Computer Engineering Group

ADVANCED JAVA PROGRAMMING

T. B. Kute

Advanced Java Programming

(Sub Code: 9165)
Computer Engineering Group, Semester: VI

First Edition
(March 2009)

T. B. Kute
(B. E. Computer)
Lecturer in Information Technology,
K. K. Wagh Polytechnic,
Nashik, India, 422003.

Windows of Knowledge are Wide and Open

Dedicated to All the Learners…….

who are keen to learn some exciting features of Java

Syllabus Structure Given by Maharashtra State Board of Technical Education, Mumbai as per revised syllabus introduced from academic year 2006-2007.

Course Name: Computer Engineering Group **Course code :** CO/CM/IF/CD

Semester: Sixth for **CO / CM / IF** and Seventh for **CD**

Subject Title: Advanced Java Programming **Subject Code :** 9165

Teaching and Examination Scheme:									
Teaching Scheme			Examination Scheme						
TH	TU	PR	Paper Hours	TH	TEST	PR	OR	TW	TOTAL
03	--	04	03	80	20	50#	--	25@	175

Rationale:

In the current era of networking, online transaction processing and managing the dataflow over network becomes an important issue. This subject is essential for providing knowledge and hands on experience over the issues of managing data on web, developing powerful GUI based friendly user interface, server side programming and developing applications for communication over network using object oriented fundamentals.

Advanced Java enhances the Java programming. After learning this subject, student will be able to develop network based software projects required in curriculum as well as industry

Objectives:

After studying this subject, the student will be able to:

- Create network based applications.
- Create business applications.
- Implement Server side programming.
- Develop dynamic software components.
- Develop database application.
- Design and develop powerful GUI based components.
- Create Animation using Applet, Thread and AWT controls.

Contents: Theory

Chapter	Name of the Topic	Hours	Marks
01	**Introduction the Abstract Window Toolkit (AWT)** 1.1 Working with Windows and AWT AWT classes Windows Fundamentals Working with frame windows Creating a frame window in applet Creating windowed program Display information within with in a window 1.2 Working with graphics Working with color Setting the paint mode Working with Fonts Managing text output using Font Metrics Exploring text & graphics 1.3 Using AWT Controls, Layout Managers and Menus Control Fundamentals Labels Using Buttons Applying Check Boxes Checkbox Group Choice Controls Using Lists Managing scroll Bars Using a Text Field Using a Text Area Understanding Layout Managers Menu Bars and Menu Dialog Boxes File Dialog Handling events by Extending AWT Components Exploring the Controls, Menus, and Layout Managers	16	20
02	**Networking** 2.1 Basics Socket overview, client/server, reserved sockets ,proxy servers, internet addressing. 2.2 Java & the Net The networking classes & interfaces 2.3 Inet address	08	16

	Factory methods, instance method 2.4 TCP/IP Client Sockets What is URL Format 2.5 URL connection 2.6 TCI/IP Server Sockets 2.7 Data grams Data gram packets, Data gram server & client		
03	**Java Data Base Client / Server** 3.1 Java as a Database front end Database client/server methodology Two-Tier Database Design Three-Tier Database Design 3.2 The JDBC API The API Components, Limitations Using JDBC(Applications vs. Applets) , Security Considerations , A JDBC Database Example JDBC Drivers ,JDBC-ODBC Bridge Current JDBC Drivers	08	14
04	**The Tour of Swing** 4.1 J applet, Icons and Labels ,Text Fields, Buttons, Combo Boxes, Tabbed Panes , Scroll Panes, 4.2 Trees, Tables, Exploring the Swings	08	14
05	**Servlets** 5.1 Background, The Life Cycle Of a Servlet, The Java Servlet Development Kit, The Simple Servlet, The Servlet API 5.2 The javax.servlet Package, Reading Servlet Parameters Reading Initialization Parameters The javax.servlet.http package, Handling HTTP Requests and responses 5.3 Using Cookies, Session Tracking, Security Issues, Exploring Servlet	08	16
	Total	**48**	**80**

Chapter 01
Introduction the Abstract Window Toolkit (AWT)

Lectures allotted: 16
Marks Given: 20

Contents:

1.1 Working with Windows and AWT
AWT classes
Windows Fundamentals
Working with frame windows
Creating a frame window in applet
Creating windowed program
Display information within a window
1.2 Working with graphics
Working with color
Setting the paint mode
Working with Fonts
Managing text output using Font Metrics
Exploring text & graphics
1.3 Using AWT Controls, Layout Managers and Menus
Control Fundamentals
Labels
Using Buttons
Applying Check Boxes
Checkbox Group
Choice Controls
Using Lists
Managing scroll Bars
Using a Text Field
Using a Text Area
Understanding Layout Managers
Menu Bars and Menu
Dialog Boxes
File Dialog
1.4 Handling events by Extending AWT Components
Exploring the Controls, Menus, and Layout Managers

Abstract Window Toolkit

The AWT contains numerous classes and methods that allow us to create and manage windows. Although the main purpose of the AWT is to support applet windows, it can also be used to create stand-alone windows that run in a GUI environment, such as Windows.

AWT Classes

The AWT classes are contained in the java.awt package. It is one of Java's largest packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use than you might at first believe.

AWTEvent	Encapsulates AWT events.
AWTEventMulticaster	Dispatches events to multiple listeners.
BorderLayout	The border layout manager. Border layouts use five components: North, South, East, West, and Center.
Button	Creates a push button control.
Canvas	A blank, semantics-free window.
CardLayout	The card layout manager. Card layouts emulate index cards. Only the one on top is showing.
Checkbox	Creates a check box control.
CheckboxGroup	Creates a group of check box controls.
CheckboxMenuItem	Creates an on/off menu item.
Choice	Creates a pop-up list.
Color	Manages colors in a portable, platform-independent fashion.
Component	An abstract super-class for various AWT components.
Container	A subclass of Component that can hold other components.
Cursor	Encapsulates a bitmapped cursor.
Dialog	Creates a dialog window.
Dimension	Specifies the dimensions of an object. The width is stored in ***width***, and the ***height*** is stored in height.
Event	Encapsulates events.
EventQueue	Queues events.
FileDialog	Creates a window from which a file can be selected.
FlowLayout	The flow layout manager. Flow layout positions components left to right, top to bottom.
Font	Encapsulates a type font.
FontMetrics	Encapsulates various information related to a font. This information helps you display text in a window.
Frame	Creates a standard window that has a title bar, resize corners, and a menu bar.

Graphics	Encapsulates the graphics context. This context is used by the various output methods to display output in a window.
GraphicsDevice	Describes a graphics device such as a screen or printer.
GraphicsEnvironment	Describes the collection of available Font and GraphicsDevice objects.
GridBagConstraints	Defines various constraints relating to the GridBagLayout class.
GridBagLayout	The grid bag layout manager. Grid bag layout displays components subject to the constraints specified by GridBagConstraints.
GridLayout	The grid layout manager. Grid layout displays components in a two-dimensional grid.
Image	Encapsulates graphical images.
Insets	Encapsulates the borders of a container.
Label	Creates a label that displays a string.
List	Creates a list from which the user can choose. Similar to the standard Windows list box.
MediaTracker	Manages media objects.
Menu	Creates a pull-down menu.
MenuBar	Creates a menu bar.
MenuComponent	An abstract class implemented by various menu classes.
MenuItem	Creates a menu item.
MenuShortcut	Encapsulates a keyboard shortcut for a menu item.
Panel	The simplest concrete subclass of Container.
Point	Encapsulates a Cartesian coordinate pair, stored in ***x*** and ***y***.
Polygon	Encapsulates a polygon.
PopupMenu	Encapsulates a pop-up menu.
PrintJob	An abstract class that represents a print job.
Rectangle	Encapsulates a rectangle.
Robot	Supports automated testing of AWT- based applications.
Scrollbar	Creates a scroll bar control.
ScrollPane	A container that provides horizontal and/or vertical scroll bars for another component.
SystemColor	Contains the colors of GUI widgets such as windows, scroll bars, text, and others.
TextArea	Creates a multiline edit control.
TextComponent	A superclass for TextArea and TextField.
TextField	Creates a single-line edit control.
Toolkit	Abstract class implemented by the AWT.
Window	Creates a window with no frame, no menu bar,

Window Fundamentals

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common windows are those derived from Panel, which is used by applets, and those derived from Frame, which creates a standard window. Much of the functionality of these windows is derived from their parent classes. Thus, a description of the class hierarchies relating to these two classes is fundamental to their understanding. Figure below shows the class hierarchy for Panel and Frame.

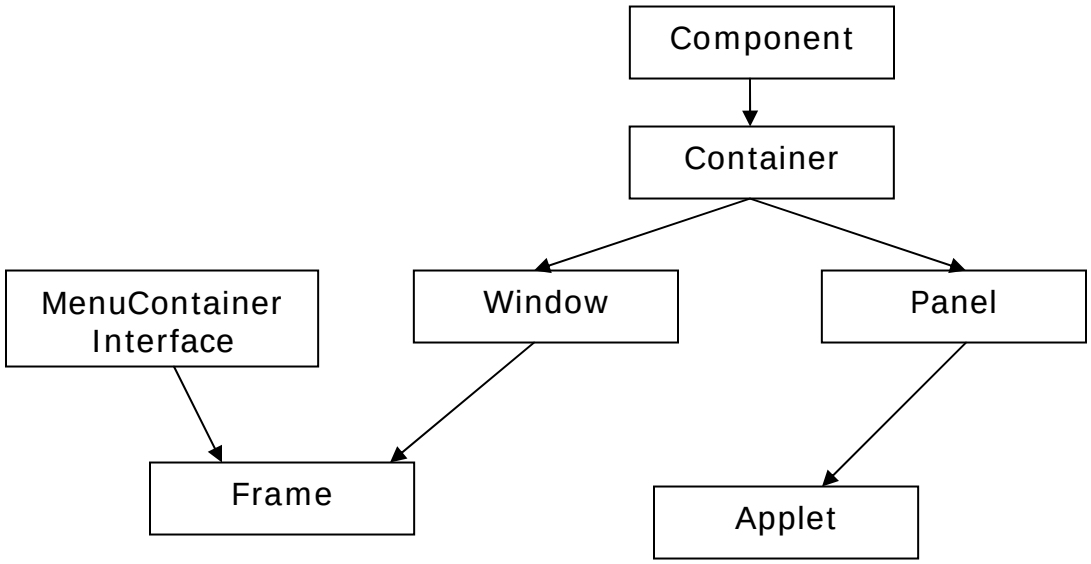


Fig. Applet class hierarchy

Component

At the top of the AWT hierarchy is the Component class. Component is an abstract class that encapsulates all of the attributes of a **visual component**. **All user interface elements that are displayed on the screen and that interact with the user are subclasses of Component.** It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting. A Component object is responsible for remembering the current foreground and background **colors** and the currently selected text **font**.

Container

The Container class is a subclass of Component. It has additional methods that allow other Component objects to be nested within it. Other Container objects can be stored inside of a Container (since they are themselves instances of Component). This makes for a multileveled containment system. **A container is responsible for laying out (that is,**

positioning) any components that it contains. It does this through the use of various layout managers.

Panel

The Panel class is a concrete subclass of Container. It doesn't add any new methods; *it simply implements Container*. A Panel may be thought of as a recursively nestable, concrete screen component. Panel is the super-class for Applet. When screen output is directed to an applet, it is drawn on the surface of a Panel object. In essence, **a Panel is a window that does not contain a title bar, menu bar, or border**. This is why we don't see these items when an applet is run inside a browser. When we run an applet using an applet viewer, the applet viewer provides the title and border. Other components can be added to a Panel object by its add( ) method (inherited from Container). Once these components have been added, we can position and resize them manually using the setLocation( ), setSize( ), or setBounds() methods defined by Component.

Window

The Window class creates a top-level window. A top-level window is not contained within any other object; it sits directly on the desktop. Generally, we won't create Window objects directly. Instead, we will use a subclass of Window called Frame.

Frame

Frame encapsulates what is commonly thought of as a "window." **It is a subclass of Window and has a title bar, menu bar, borders, and resizing corners.** If we create a Frame object from within an applet, it will contain a warning message, such as "Java Applet Window," to the user that an applet window has been created. This message warns users that the window they see was started by an applet and not by software running on their computer. When a Frame window is created by a program rather than an applet, a normal window is created.

Canvas

Canvas encapsulates a blank window upon which we can draw.

Dimension

This class encapsulates the 'width' and 'height' of a component (in integer precision) in a single object. The class is associated with certain properties of components. Several methods defined by the Component class and the LayoutManager interface return a Dimension object.

Normally the values of width and height are non-negative integers. The constructors that allow us to create a dimension do not prevent us from setting a negative value for these properties. If the value of width or height is negative, the behavior of some methods defined by other objects is undefined.

Fields of Dimension:

int height	The height dimension; negative values can be used.
int width	The width dimension; negative values can be used.

Constructors:

Dimension()
It creates an instance of Dimension with a width of zero and a height of zero.

Dimension(Dimension d)
It creates an instance of Dimension whose width and height are the same as for the specified dimension.

Dimension(int width, int height)
It constructs a Dimension and initializes it to the specified width and specified height.

Methods:

`boolean equals(Object obj)`
It checks whether two dimension objects have equal values.

`double getHeight()`
It returns the height of this dimension in double precision.

`Dimension getSize()`
It gets the size of this Dimension object.

`double getWidth()`
It returns the width of this dimension in double precision.

`void setSize(Dimension d)`
It sets the size of this Dimension object to the specified size.

`void setSize(double width, double height)`
It sets the size of this Dimension object to the specified width and height in double precision.

`void setSize(int width, int height)`
It sets the size of this Dimension object to the specified width and height.

Working with Frame Windows

After the applet, the type of window we will most often create is derived from Frame. We will use it to create child windows within applets, and top-level or child windows for applications. It creates a standard-style window. Following are two of Frame's constructors:

```
Frame( )
Frame(String title)
```

The first form creates a standard window that does not contain a title. The second form creates a window with the title specified by title. Note that we cannot specify the dimensions of the window. Instead, we must set the size of the window after it has been created.

Setting the Windows Dimensions

The setSize( ) method is used to set the dimensions of the window. Its signature is shown below:

```
void setSize(int newWidth, int newHeight)
void setSize(Dimension newSize)
```

The new size of the window is specified by 'newWidth' and 'newHeight', or by the 'width' and 'height' fields of the Dimension object passed in 'newSize'. The dimensions are specified in terms of pixels. The getSize( ) method is used to obtain the current size of a window. Its signature is:

```
Dimension getSize( )
```

This method returns the current size of the window contained within the 'width' and 'height' fields of a Dimension object.

Hiding and showing a Window

After a frame window has been created, it will not be visible until we call setVisible( ). Its signature is:

```
void setVisible(boolean visibleFlag)
```

The component is visible if the argument to this method is true. Otherwise, it is hidden.

Setting a Windows Title

We can change the title in a frame window using setTitle( ), which has this general form:

```
void setTitle(String newTitle)
```

Here, 'newTitle' is the new title for the window.

Closing a Frame Window

When using a frame window, our program must remove that window from the screen when it is closed, by calling setVisible(false). To intercept a window-close event, we must implement the windowClosing( ) method of the WindowListener interface. Inside windowClosing( ), we must remove the window from the screen.

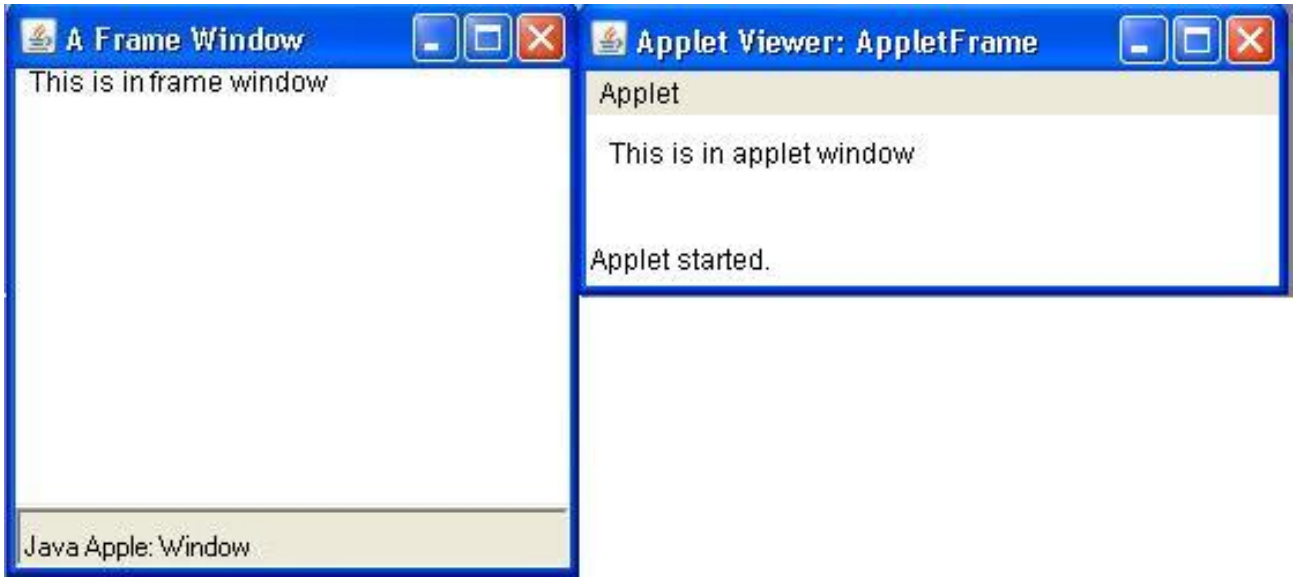
Creating a Frame Window in an Applet

The following applet creates a subclass of Frame called SampleFrame. A window of this subclass is instantiated within the init( ) method of AppletFrame. Note that 'SampleFrame' calls Frame's constructor. This causes a standard frame window to be created with the title passed in title. This example overrides the applet window's start( ) and stop( ) method s so that they show and hide the child window, respectively. It also causes the child window to be shown when the browser returns to the applet.

```
/*
   <applet code="AppletFrame" width=300 height=50>
   </applet>
*/
class SampleFrame extends Frame
{
        SampleFrame(String title)
        {
             super(title);
        }
        public void paint(Graphics g)
        {
             g.drawString("This is in frame window", 10, 40);
        }
}
public class AppletFrame extends Applet
{
        Frame f;
        public void init()
        {
             f = new SampleFrame("A Frame Window");
             f.setSize(250, 250);
```

```
            f.setVisible(true);
        }
        public void start()
        {
            f.setVisible(true);
        }
        public void stop()
        {
            f.setVisible(false);
        }
        public void paint(Graphics g)
        {
            g.drawString("This is in applet window", 10, 20);
        }
}
```

Output:



Creating a Windowed Program

Although creating applets is the most common use for Java's AWT, it is possible to create stand-alone AWT-based applications, too. To do this, simply create an instance of the window or windows we need inside main( ). For example, the following program creates a simple frame window.

```
public class AppWindow extends Frame
{
    AppWindow(String title)
    {
       super(title);
```

```
        }
        public void paint(Graphics g)
        {
           setForeground(Color.red);
           setBackground(Color.cyan);
           g.drawString("This is my frame", 30, 70);
        }
        public static void main(String args[])
           throws Exception
        {
           AppWindow appwin = new AppWindow("Frame window");
           appwin.setSize(new Dimension(300, 200));
           appwin.setVisible(true);
           Thread.sleep(5000);
           appwin.setTitle("An AWT-Based Application");
           Thread.sleep(5000);
           System.exit(0);
        }
   }
```

Another Program:

```
   public class AppWindow extends Frame
   {
        Font f;
        static int val=5;
        AppWindow(String title)
        {
           super(title);
        }
        public void paint(Graphics g)
        {
           setForeground(Color.red);
           setBackground(Color.cyan);

           Integer i = new Integer(val);
           g.drawString(i.toString(), 30, 70);
           val--;
        }
        public static void main(String args[])
           throws Exception
        {
           AppWindow appwin = new AppWindow("Frame window");
           appwin.setSize(new Dimension(300, 200));
           appwin.setFont(new Font("Arial",Font.BOLD, 40));
           appwin.setVisible(true);
           Thread.sleep(5000);
           appwin.setTitle("An AWT-Based Application");
```

```
            for(int i=0;i<5;i++)
            {
               Thread.sleep(1000);
               appwin.repaint();
            }
            System.exit(0);
         }
   }
```

Displaying information within a Window

In the most general sense, a window is a container for information. Although we have already output small amounts of text to a window in the preceding examples, we have not begun to take advantage of a window's ability to present high-quality text and graphics. Indeed, much of the power of the AWT comes from its support for these items.

Working with Graphics

Working with Color

Java supports color in a portable, device-independent fashion. The AWT color system allows us to specify any color that we want. It then finds the best match for that color, given the limits of the display hardware currently executing our program or applet. Thus, our code does not need to be concerned with the differences in the way color is supported by various hardware devices. Color is encapsulated by the Color class.

Constructors:

```
Color(int red, int green, int blue)
Color(int rgbValue)
Color(float red, float green, float blue)
```

The first constructor takes three integers that specify the color as a mix of red, green, and blue. These values must be between 0 and 255, as in this example:

```
new Color(255, 100, 100); // light red.
```

The second color constructor takes a single integer that contains the mix of red, green, and blue packed into an integer. The integer is organized with red in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7. Here is an example of this constructor:

```
int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);
```

Color darkRed = new Color(newRed);

The final constructor, Color(float, float, float), takes three float values (between 0.0 and 1.0) that specify the relative mix of red, green, and blue. Once we have created a color, we can use it to set the foreground and/or background color by using the setForeground( ) and setBackground( ) methods. You can also select it as the current drawing color.

Color Methods

The Color class defines several methods that help manipulate colors.

Using Hue, Saturation, and Brightness

The hue-saturation-brightness (HSB) color model is an alternative to red-green-blue (RGB) for specifying particular colors. Figuratively, hue is a wheel of color. The hue is specified with a number between 0.0 and 1.0 (the colors are approximately: red, orange, yellow, green, blue, indigo, and violet). Saturation is another scale ranging from 0.0 to 1.0, representing light pastels to intense hues. Brightness values also range from 0.0 to 1.0, where 1 is bright white and 0 is black. Color supplies two methods that let us convert between RGB and HSB. They are shown here:

```
static int HSBtoRGB(float hue, float sat, float brightness)
static float[ ] RGBtoHSB(int r, int g, int b, float
          values[])
```

HSBtoRGB( ) returns a packed RGB value compatible with the Color(int) constructor. RGBtoHSB( ) returns a float array of HSB values corresponding to RGB integers. If *values* is not null, then this array is given the HSB values and returned. Otherwise, a new array is created and the HSB values are returned in it. In either case, the array contains the hue at index 0, saturation at index 1, and brightness at index 2.

getRed( ), getGreen( ), getBlue( )

We can obtain the red, green, and blue components of a color independently using getRed( ), getGreen( ), and getBlue( ), shown below:

```
int getRed( )
int getGreen( )
int getBlue( )
```

Each of these methods returns the RGB color component found in the invoking Color object in the lower 8 bits of an integer.

getRGB( )

To obtain a packed, RGB representation of a color, use getRGB( ), shown here:

```
int getRGB( )
```

The return value is organized as described earlier.

Setting the Current Graphics Color

By default, graphics objects are drawn in the current foreground color. We can change this color by calling the Graphics method setColor( ):

```
void setColor(Color newColor)
```

Here, 'newColor' specifies the new drawing color. We can obtain the current color by calling getColor( ), shown below:

```
Color getColor( )
```

Example:

```
/*
     <applet code="ColorDemo" width=300 height=200>
     </applet>
*/
public class ColorDemo extends Applet
{
     public void paint(Graphics g)
     {
        Color c1 = new Color(202, 146, 20);
        Color c2 = new Color(110, 169, 107);
        Color c3 = new Color(160, 100, 200);

        g.setColor(c1);
        g.drawLine(0, 0, 100, 100);
        g.drawLine(0, 100, 100, 0);

        g.setColor(Color.red);
        g.drawLine(40, 25, 250, 180);

        g.setColor(c3);
        g.drawLine(20, 150, 400, 40);

        g.setColor(c2);
        g.drawOval(10, 10, 50, 50);
```

```
            g.fillOval(70, 90, 140, 100);
        }
    }
```

Setting the Paint Mode

The paint-mode determines how objects are drawn in a window. By default, new output to a window overwrites any pre-existing contents. However, it is possible to have new objects XORed onto the window by using setXORMode( ), as follows:

```
void setXORMode(Color xorColor)
```

Here, 'xorColor' specifies the color that will be XORed to the window when an object is drawn. The advantage of XOR mode is that the new object is always guaranteed to be visible no matter what color the object is drawn over. To return to overwrite mode, call setPaintMode( ), shown here:

```
void setPaintMode( )
```

In general, we will want to use overwrite mode for normal output, and XOR mode for special purposes. For example, the following program displays cross hairs that track the mouse pointer. The cross hairs are XORed onto the window and are always visible, no matter what the underlying color is.

Working with Fonts

The AWT supports multiple type fonts. It provides flexibility by abstracting font-manipulation operations and allowing for dynamic selection of fonts. In Java, fonts have a family name, a logical font name, and a face name. The family name is the general name of the font, such as Courier. The logical name specifies a category of font, such as Monospaced. The face name specifies a specific font, such as Courier Italic. Fonts are encapsulated by the Font class. Several of the methods defined by Font are listed below.

The Font class defines these variables:

Variable	Meaning
String name	Name of the font
float pointSize	Size of the font in points
int size	Size of the font in points
int style	Font style

Method	Description
static Font decode(String str)	Returns a font given its name.
boolean equals(Object FontObj)	Returns true if the invoking object contains the same font as that specified by FontObj.

	Otherwise, it returns false.
String getFamily( )	Returns the name of the font family to which the invoking font belongs.
staticFontgetFont(Stringproperty)	Returns the font associated with the system property specified by property. Null is returned if property does not exist.
staticFontgetFont(Stringproperty, Font defaultFont)	Returns the font associated with the system property specified by property. The font specified by default Font is returned if property does not exist.
String getFontName()	Returns the face name of the invoking font.
String getName( )	Returns the logical name of the invoking font.
int getSize( )	Returns the size, in points, of the invoking font.
int getStyle( )	Returns the style values of the invoking font.
int hashCode( )	Returns the hash code associated with the invoking object.
boolean isBold( )	Returns true if the font includes the BOLD style value. Otherwise, false is returned.
boolean isItalic( )	Returns true if the font includes the ITALIC style value. Otherwise, false is returned.
boolean isPlain( )	Returns true if the font includes the PLAIN style value. Otherwise, false is returned.
String toString( )	Returns the string equivalent of the invoking font.

Determining the Available Fonts

When working with fonts, often we need to know which fonts are available on our machine. To obtain this information, we can use the getAvailableFontFamilyNames( ) method defined by the GraphicsEnvironment class.

It is shown here:

```
String[ ] getAvailableFontFamilyNames( )
```

This method returns an array of strings that contains the names of the available font families. In addition, the getAllFonts( ) method is defined by the GraphicsEnvironment class. It is shown here:

```
Font[ ] getAllFonts( )
```

This method returns an array of Font objects for all of the available fonts. Since these methods are members of GraphicsEnvironment, we need a GraphicsEnvironment reference to call them. We can obtain this reference by using the getLocalGraphicsEnvironment( ) static method, which is defined by GraphicsEnvironment. It is shown here:

```
static GraphicsEnvironment getLocalGraphicsEnvironment( )
```

Here is an applet that shows how to obtain the names of the available font families:

```
/*
 <applet code="ShowFonts" width=550 height=60>
 </applet>
 */
 import java.applet.*;
 import java.awt.*;

 public class ShowFonts extends Applet
 {
      public void paint(Graphics g)
      {
          String msg = "";
          String FontList[];

          GraphicsEnvironment ge =
              GraphicsEnvironment.getLocalGraphicsEnvironment();
          FontList = ge.getAvailableFontFamilyNames();
          for(int i = 0; i < FontList.length; i++)
              msg += FontList[i] + " ";

          g.drawString(msg, 4, 16);
      }
 }
```

Creating and Selecting a Font

In order to select a new font, we must first construct a Font object that describes that font. One Font constructor has this general form:

```
Font(String fontName, int fontStyle, int pointSize)
```

Here, 'fontName' specifies the name of the desired font. The name can be specified using either the logical or face name. All Java environments will support the following fonts:

- Dialog,

- DialogInput,
- Sans Serif,
- Serif,
- Monospaced
- Symbol

Dialog is the font used by our system's dialog boxes. Dialog is also the default if we don't explicitly set a font. We can also use any other fonts supported by our particular environment, but these other fonts may not be universally available. The style of the font is specified by 'fontStyle'. It may consist of one or more of these three constants:

Font.PLAIN, Font.BOLD, and Font.ITALIC.

For combining styles, we can OR them together.

For example, Font.BOLD | Font.ITALIC specifies a bold, italics style.

The size, in points, of the font is specified by 'pointSize'. For using a font that we have created, we must select it using setFont( ), which is defined by Component. It has this general form:

```
void setFont(Font fontObj)
```

Here, fontObj is the object that contains the desired font.

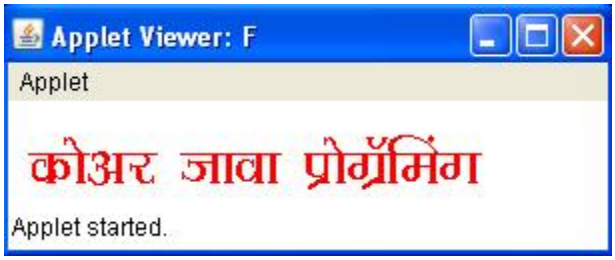
```
// Displaying different  fonts
  import java.awt.Font;
  import java.awt.Graphics;
  import java.applet.Applet;
  /*
  <applet code="Fonts" width=300 height=150>
  </applet>
  */
  public class Fonts extends Applet
  {
    public void paint(Graphics g)
    {
        Font f1 = new Font("TimesRoman", Font.PLAIN, 18);
        Font f2 = new Font("Courier", Font.BOLD, 16);
        Font f3 = new Font("Arial", Font.ITALIC, 20);
        Font f4 = new Font("Times", Font.BOLD + Font.ITALIC,
                           22);
        g.setFont(f1);
        g.drawString("Times Roman plain font: 18", 10, 30);
        g.setFont(f2);
        g.drawString("Courier bold font: 16", 10, 60);
        g.setFont(f3);
        g.drawString("Arial italic font: 20", 10, 80);
        g.setFont(f4);
        g.drawString("Times bold italic font: 22", 10, 120);
    }
```

```
}
```

We can set any font available in our 'windows\fonts' directory to the text on the applet. For example: the Devnagari font named "shusha" is freely available on internet. The following code displays applet with that font.

```
Font f = new Font("Shusha02", Font.PLAIN, 35);
g.setColor(Color.red);
g.setFont(f);
g.drawString("kaoAr jaavaa p`aoga`^imaMga", 10, 40);
```

Output window:



Obtaining Font Information

Suppose we want to obtain information about the currently selected font. To do this, we must first get the current font by calling getFont( ). This method is defined by the Graphics class, as shown here:

```
Font getFont( )
```

Once we have obtained the currently selected font, we can retrieve information about it using various methods defined by Font. For example, this applet displays the name, family, size, and style of the currently selected font:

```
// Display font info.
/*
<applet code="FontInfo" width=350 height=60>
</applet>
*/

public class FontInfo extends Applet
{
  public void paint(Graphics g)
  {
    Font f = g.getFont();
    String fontName = f.getName();
    String fontFamily = f.getFamily();
    int fontSize = f.getSize();
    int fontStyle = f.getStyle();

    String msg = "Family: " + fontName;
    msg += ", Font: " + fontFamily;
    msg += ", Size: " + fontSize + ", Style: ";
    if((fontStyle & Font.BOLD) == Font.BOLD)
      msg += "Bold ";
    if((fontStyle & Font.ITALIC) == Font.ITALIC)
      msg += "Italic ";
    if((fontStyle & Font.PLAIN) == Font.PLAIN)
      msg += "Plain ";

    g.drawString(msg, 4, 16);
  }
}
```

Managing Text Output Using FontMetrics

Java supports a number of fonts. For most fonts, characters are not all the same dimension most fonts are proportional. Also, the height of each character, the length of descenders (the hanging parts of letters, such as y), and the amount of space between horizontal lines vary from font to font. Further, the point size of a font can be changed. That these (and other) attributes are variable would not be of too much consequence except that Java demands that the programmer, manually manage virtually all text output. Given that the size of each font may differ and that fonts may be changed while our program is executing, there must be some way to determine the dimensions and various other attributes of the currently selected font. For example, to write one line of text after another implies that we have some way of knowing how tall the font is and how many pixels are needed between lines. To fill this need, the AWT includes the FontMetrics class, which encapsulates

various information about a font. Common terminology used when describing fonts:

Height	The top-to-bottom size of the tallest character in the font.
Baseline	The line that the bottoms of characters are aligned to (not counting descent).
Ascent	The distance from the baseline to the top of a character
Descent	The distance from the baseline to the bottom of a character
Leading	The distance between the bottom of one line of text and the top of the next.

We have used the drawString( ) method in many of the previous examples. It paints a string in the current font and color, beginning at a specified location. However, this location is at the left edge of the baseline of the characters, not at the upper-left corner as is usual with other drawing methods. It is a common error to draw a string at the same coordinate that we would draw a box. For example, if you were to draw a rectangle at coordinate 0,0 of your applet, you would see a full rectangle. If you were to draw the string Typesetting at 0,0, you would only see the tails (or descenders) of the y, p, and g.As you will see, by using font metrics, you can determine the proper placement of each string that you display. FontMetrics defines several methods that help you manage text output. The most commonly used are listed below. These methods help you properly display text in a window. Let's look at some examples.

`int bytesWidth(byte b[ ], int start, int numBytes)`
Returns the width of numBytes characters held in array b, beginning at start.

`int charWidth(char c[ ], int start, int numChars)`
Returns the width of numChars characters held in array c, beginning at start.

int charWidth(char c)	Returns the width of c.
int charWidth(int c)	Returns the width of c.
int getAscent( )	Returns the ascent of the font.
int getDescent( )	Returns the descent of the font.
Font getFont( )	Returns the font.
int getHeight( )	Returns the height of a line of text. This value can be used to output multiple lines of text in a window.
int getLeading( )	Returns the space between lines of text.
int getMaxAdvance( )	Returns the width of the widest character. –1 is returned if this value is not available.
int getMaxAscent( )	Returns the maximum ascent.
int getMaxDescent( )	Returns the maximum descent.
int[ ] getWidths( )	Returns the widths of the first 256 characters.
int stringWidth(String str)	Returns the width of the string specified by str.

String toString( )	Returns the string equivalent of the invoking object.

Exploring Text and Graphics

Although we have covered the most important attributes and common techniques that we will use when displaying text or graphics, it only scratches the surface of Java's capabilities. This is an area in which further refinements and enhancements are expected as Java and the computing environment continue to evolve. For example, Java 2 added a subsystem to the AWT called Java 2D. Java 2D supports enhanced control over graphics, including such things as coordinate translations, rotation, and scaling. It also provides advanced imaging features. If advanced graphics handling is of interest to us, then we will definitely want to explore Java 2D in detail.

Using AWT Controls, Layout Managers and Menus

Controls are components that allow a user to interact with his application in various ways—for example; a commonly used control is the push button. A layout manager automatically positions components within a container. Thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them. In addition to the controls, a frame window can also include a standard-style menu bar. Each entry in a menu bar activates a drop-down menu of options from which the user can choose. A menu bar is always positioned at the top of a window. Although different in appearance, menu bars are handled in much the same way as are the other controls. While it is possible to manually position components within a window, doing so is quite tedious. The layout manager automates this task.

Control Fundamentals

The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Choice lists
- Lists
- Scroll bars
- Text Area
- Text Field

These controls are subclasses of **Component**.

Adding and Removing Controls

In order to include a control in a window, we must add it to the window. So, we must first create an instance of the desired control and then add it to a window by calling add( ), which is defined by **Container**. The add( ) method has several forms. The following form is the one that is used for the first part of this chapter:

```
Component add(Component compObj)
```

Here, compObj is an instance of the control that we want to add. A reference to compObj is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed. Sometimes we will want to remove a control from a window when the control is no longer needed. For doing this, call remove( ). This method is also defined by Container. It has this general form:

```
void remove(Component obj)
```

Here, obj is a reference to the control that we want to remove. We can remove all controls by calling removeAll( ).

Responding to Controls

Except for labels, which are passive controls, all controls generate events when they are accessed by the user. For example, when the user clicks on a push button, an event is sent that identifies the push button. In general, our program simply implements the appropriate interface and then registers an event listener for each control that we need to monitor.

Labels

The easiest control to use is a label. A label is an object of type Label, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user. Label defines the following constructors:

```
Label( )
Label(String str)
Label(String str, int how)
```

The first version creates a blank label. The second version creates a label that contains the string specified by *str*. This string is left-justified. The third version creates a label that contains the string specified by *str* using the

alignment specified by *how*. The value of how must be one of these three constants: Label.LEFT, Label.RIGHT, or Label.CENTER.

We can set or change the text in a label by using the setText( ) method. We can obtain the current label by calling getText( ). These methods are shown here:

```
void setText(String str)
String getText( )
```

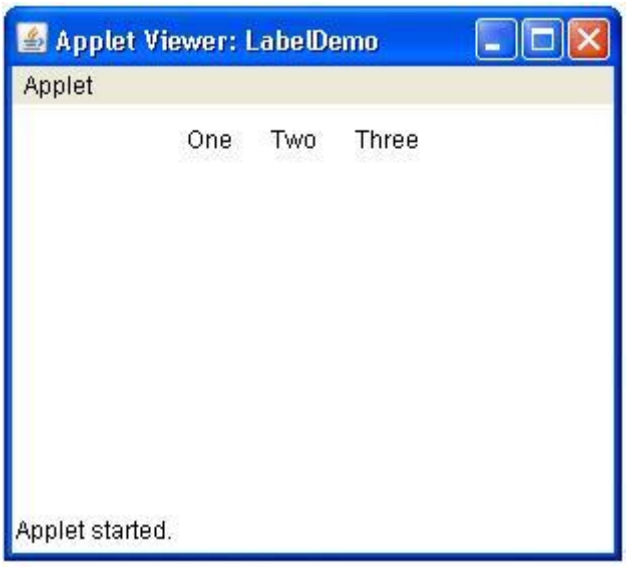
For setText( ), str specifies the new label. For getText( ), the current label is returned. You can set the alignment of the string within the label by calling setAlignment( ). To obtain the current alignment, call getAlignment( ). The methods are as follows:

```
void setAlignment(int how)
int getAlignment( )
```

Here, how must be one of the alignment constants shown earlier. The following example creates three labels and adds them to an applet:

```
// Demonstrate Labels
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/
public class LabelDemo extends Applet
{
     public void init()
     {
          Label one = new Label("One");
          Label two = new Label("Two");
          Label three = new Label("Three");
          // add labels to applet window
          add(one);
          add(two);
          add(three);
     }
}
```

Following is the window created by the LabelDemo applet. Notice that the labels are organized in the window by the default layout manager.



Buttons

The most widely used control is the push button. A push button is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type Button. Button defines these two constructors:

```
Button( )
Button(String str)
```

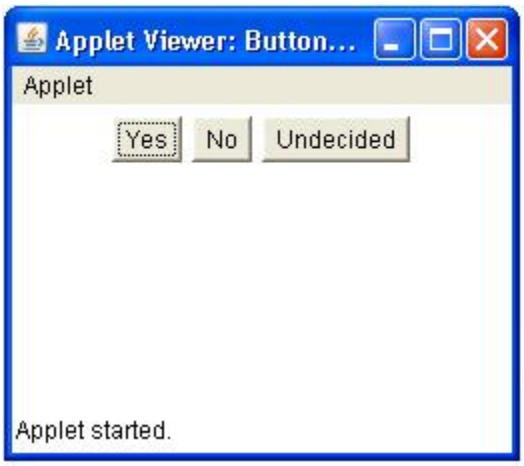
The first version creates an empty button. The second creates a button that contains *str* as a label. After a button has been created, we can set its label by calling setLabel( ). We can retrieve its label by calling getLabel( ). These methods are as follows:

```
void setLabel(String str)
String getLabel( )
```

Here, *str* becomes the new label for the button.

```
// Demonstrate Buttons
import java.awt.*;
import java.applet.*;
/*
<applet code="ButtonDemo" width=250 height=150>
</applet>
*/
public class ButtonDemo extends Applet
{
```

```
        String msg = "";
        Button yes, no, maybe;
        public void init()
        {
                yes = new Button("Yes");
                no = new Button("No");
                maybe = new Button("Undecided");
                add(yes);
                add(no);
                add(maybe);
        }
        public void paint(Graphics g)
        {
                g.drawString(msg, 6, 100);
        }
}
```



Check Boxes

A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. We change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the Checkbox class. Checkbox supports these constructors:

```
Checkbox( )
Checkbox(String str)
Checkbox(String str, boolean on)
Checkbox(String str, boolean on, CheckboxGroup cbGroup)
Checkbox(String str, CheckboxGroup cbGroup, boolean on)
```

The first form creates a check box whose label is initially blank. The state of the check box is unchecked. The second form creates a check box whose label is specified by *str*. The state of the check box is unchecked. The third form allows us to set the initial state of the check box. If on is true, the check box is initially checked; otherwise, it is cleared. The fourth and fifth forms create a check box whose label is specified by *str* and whose group is specified by *cbGroup*. If this check box is not part of a group, then *cbGroup* must be null. The value of on determines the initial state of the check box.

In order to retrieve the current state of a check box, call getState( ). For setting its state, call setState( ). We can obtain the current label associated with a check box by calling getLabel( ). For setting the label, setLabel( ) is used. These methods are as follows:

```
boolean getState( )
void setState(boolean on)
String getLabel( )
void setLabel(String str)
```

Here, if on is true, the box is checked. If it is false, the box is cleared. The string passed in *str* becomes the new label associated with the invoking check box.

```
// Demonstrate check boxes.
import java.awt.*;
import java.applet.*;
/*
<applet code="CheckboxDemo" width=250 height=200>
</applet>
*/
public class CheckboxDemo extends Applet
{
    String msg = "";
    Checkbox Win98, winNT, solaris, mac;
    public void init()
    {
        Win98 = new Checkbox("Windows 98/XP", null, true);
        winNT = new Checkbox("Windows NT/2000");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("MacOS");
        add(Win98);
        add(winNT);
        add(solaris);
        add(mac);
    }
    public void paint(Graphics g)
    {
    }
```

```
}
```

Checkbox Group

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called radio buttons, because they act like the station selector on a car radio—only one station can be selected at any one time. For creating a set of mutually exclusive check boxes, we must first define the group to which they will belong and then specify that group when we construct the check boxes. Check box groups are objects of type CheckboxGroup. Only the default constructor is defined, which creates an empty group.

We can determine which check box in a group is currently selected by calling getSelectedCheckbox( ). We can set a check box by calling setSelectedCheckbox( ). These methods are as follows:

```
Checkbox getSelectedCheckbox( )
void setSelectedCheckbox(Checkbox wh)
```

Here, *wh* is the check box that we want to be selected. The previously selected check box will be turned off. Here is a program that uses check boxes that are part of a group:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="CBGroup" width=250 height=200>
</applet>
```

```
*/
public class CBGroup extends Applet
{
    String msg = "";
    Checkbox Win98, winNT, solaris, mac;
    CheckboxGroup cbg;
    public void init()
    {
        cbg = new CheckboxGroup();
        Win98 = new Checkbox("Windows 98/XP", cbg, true);
        winNT = new Checkbox("Windows NT/2000", cbg, false);
        solaris = new Checkbox("Solaris", cbg, false);
        mac = new Checkbox("MacOS", cbg, false);
        add(Win98);
        add(winNT);
        add(solaris);
        add(mac);
    }
    public void paint(Graphics g)
    {
        msg = "Current selection: ";
        msg += cbg.getSelectedCheckbox().getLabel();
        g.drawString(msg, 6, 100);
    }
}
```

Choice Controls

The Choice class is used to create a pop-up list of items from which the user may choose. Thus, a Choice control is a form of menu. When inactive, a Choice component takes up only enough space to show the currently selected

item. When the user clicks on it, the whole list of choices pops up, and a new selection can be made. Each item in the list is a string that appears as a left-justified label in the order it is added to the Choice object. Choice only defines the default constructor, which creates an empty list. In order to add a selection to the list, add( ) is used. It has this general form:

```
void add(String name)
```

Here, name is the name of the item being added. Items are added to the list in the order in which calls to add( ) occur. In order to determine which item is currently selected, we may call either any of the following methods:

```
String getSelectedItem( )
int getSelectedIndex( )
```

The getSelectedItem( ) method returns a string containing the name of the item. getSelectedIndex( ) returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected. For obtaining the number of items in the list, call getItemCount( ). We can set the currently selected item using the select( ) method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:

```
int getItemCount( )
void select(int index)
void select(String name)
```

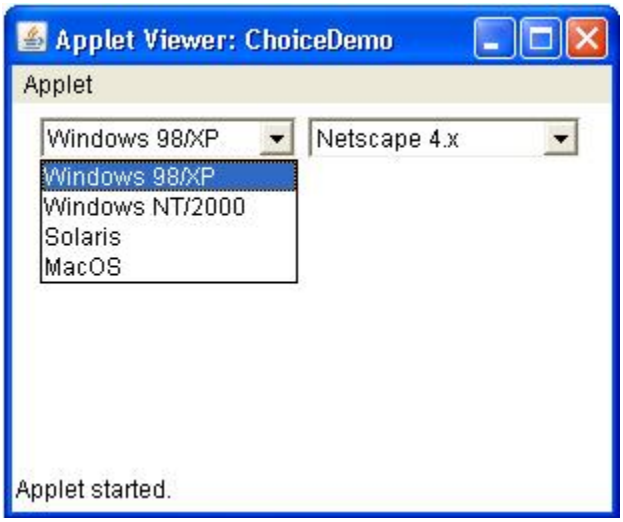
Given an index, we can obtain the name associated with the item at that index by calling getItem( ), which has this general form:

```
String getItem(int index)
```

Here, *index* specifies the index of the desired item.

```
// Demonstrate Choice lists.
import java.awt.*;
import java.applet.*;
/*
<applet code="ChoiceDemo" width=300 height=180>
</applet>
*/
public class ChoiceDemo extends Applet
{
    Choice os, browser;
    String msg = "";
    public void init()
    {
        os = new Choice();
```

```
        browser = new Choice();
        os.add("Windows 98/XP");
        os.add("Windows NT/2000");
        os.add("Solaris");
        os.add("MacOS");
        browser.add("Netscape 3.x");
        browser.add("Netscape 4.x");
        browser.add("Netscape 5.x");
        browser.add("Netscape 6.x");
        browser.add("Internet Explorer 4.0");
        browser.add("Internet Explorer 5.0");
        browser.add("Internet Explorer 6.0");
        browser.add("Lynx 2.4");
        browser.select("Netscape 4.x");
        add(os);
        add(browser);
    }
    public void paint(Graphics g)
    {
    }
}
```



Lists

The List class provides a compact, multiple-choice, scrolling selection list. Unlike the Choice object, which shows only the single selected item in the menu, a List object can be constructed to show any number of choices in the visible Window. It can also be created to allow multiple selections. List provides these constructors:

```
List( )
List(int numRows)
List(int numRows, boolean multipleSelect)
```

The first version creates a List control that allows only one item to be selected at any one time. In the second form, the value of *numRows* specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed). In the third form, if *multipleSelect* is true, then the user may select two or more items at a time. If it is false, then only one item may be selected. For adding a selection to the list, we can call add( ). It has the following two forms:

```
void add(String name)
void add(String name, int index)
```

Here, *name* is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by *index*. Indexing begins at zero. We can specify –1 to add the item to the end of the list. For lists that allow only single selection, we can determine which item is currently selected by calling either getSelectedItem( ) or getSelectedIndex( ). These methods are shown here:

```
String getSelectedItem( )
int getSelectedIndex( )
```

The getSelectedItem( ) method returns a string containing the name of the item. If more than one item is selected or if no selection has yet been made, null is returned. getSelectedIndex( ) returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, –1 is returned. For lists that allow multiple selection, we must use either getSelectedItems( ) or getSelectedIndexes( ), shown here, to determine the current selections:

```
String[ ] getSelectedItems( )
int[ ] getSelectedIndexes( )
```

The getSelectedItems( ) returns an array containing the names of the currently selected items. getSelectedIndexes( ) returns an array containing the indexes of the currently selected items. In order to obtain the number of items in the list, call getItemCount( ). We can set the currently selected item by using the select( ) method with a zero-based integer index. These methods are shown here:

```
int getItemCount( )
void select(int index)
```

Given an *index*, we can obtain the name associated with the item at that index by calling getItem( ), which has this general form:

String getItem(int index)

Here, *index* specifies the index of the desired item.

```
import java.awt.*;
import java.applet.*;
/*
<applet code="ListDemo" width=300 height=180>
</applet>
*/
public class ListDemo extends Applet
{
    List os, browser;
    String msg = "";
    public void init()
    {
        os = new List(4, true);
        browser = new List(4, false);
        os.add("Windows 98/XP");
        os.add("Windows NT/2000");
        os.add("Solaris");
        os.add("MacOS");
        browser.add("Netscape 3.x");
        browser.add("Netscape 4.x");
        browser.add("Netscape 5.x");
        browser.add("Netscape 6.x");
        browser.add("Internet Explorer 4.0");
        browser.add("Internet Explorer 5.0");
        browser.add("Internet Explorer 6.0");
        browser.add("Lynx 2.4");
        browser.select(1);
        add(os);
        add(browser);
    }
    public void paint(Graphics g)
    {
        int idx[];
        msg = "Current OS: ";
        idx = os.getSelectedIndexes();
        for(int i=0; i<idx.length; i++)
            msg += os.getItem(idx[i]) + " ";
        g.drawString(msg, 6, 120);
        msg = "Current Browser: ";
        msg += browser.getSelectedItem();
        g.drawString(msg, 6, 140);
    }
}
```

Scroll Bars

Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that we can click to move the current value of the scroll bar one unit in the direction of the arrow. The current value of the scroll bar relative to its minimum and maximum values is indicated by the slider box (or thumb) for the scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this action translates into some form of page up and page down. Scroll bars are encapsulated by the **Scrollbar** class.

Scrollbar defines the following constructors:

```
Scrollbar( )
Scrollbar(int style)
Scrollbar(int style, int iValue, int tSize, int min, int max)
```

The first form creates a vertical scroll bar. The second and third forms allow us to specify the orientation of the scroll bar. If *style* is Scrollbar.VERTICAL, a vertical scroll bar is created. If *style* is Scrollbar.HORIZONTAL, the scroll bar is horizontal. In the third form of the constructor, the initial value of the scroll bar is passed in *iValue*. The number of units represented by the height of the thumb is passed in *tSize*. The minimum and maximum values for the scroll bar are specified by min and max. If we construct a scroll bar by using one of the first two constructors, then we need to set its parameters by using setValues( ), shown here, before it can be used:

```
void setValues(int iValue, int tSize, int min, int max)
```

The parameters have the same meaning as they have in the third constructor just described. In order to obtain the current value of the scroll bar, call getValue( ). It returns the current setting. For setting the current value, we can use setValue( ). These methods are as follows:

```
int getValue( )
void setValue(int newValue)
```

Here, *newValue* specifies the new value for the scroll bar. When we set a value, the slider box inside the scroll bar will be positioned to reflect the new value. We can also retrieve the minimum and maximum values via getMinimum( ) and getMaximum( ), shown here:

```
int getMinimum( )
int getMaximum( )
```

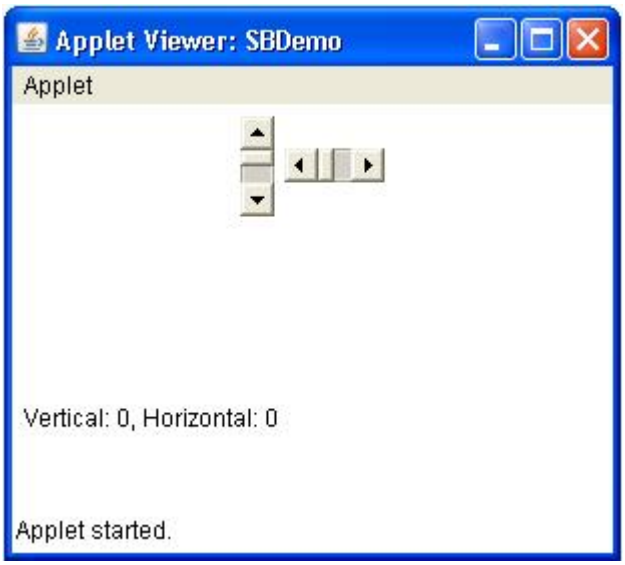
They return the requested quantity. By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line. We can change this increment by calling setUnitIncrement( ). By default, page-up and page-down increments are 10. You can change this value by calling setBlockIncrement( ). These methods are shown here:

```
void setUnitIncrement(int newIncr)
void setBlockIncrement(int newIncr)
```

Example:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SBDemo" width=300 height=200>
</applet>
*/
public class SBDemo extends Applet
{
    String msg = "";
    Scrollbar vertSB, horzSB;
    public void init()
    {
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));
        vertSB = new Scrollbar(Scrollbar.VERTICAL,
                0, 1, 0, height);
        horzSB = new Scrollbar(Scrollbar.HORIZONTAL,
                0, 1, 0, width);
        add(vertSB);
```

```
        add(horzSB);
    }
    public void paint(Graphics g)
    {
        msg = "Vertical: " + vertSB.getValue();
        msg += ", Horizontal: " + horzSB.getValue();
        g.drawString(msg, 6, 160);
    }
}
```



TextField

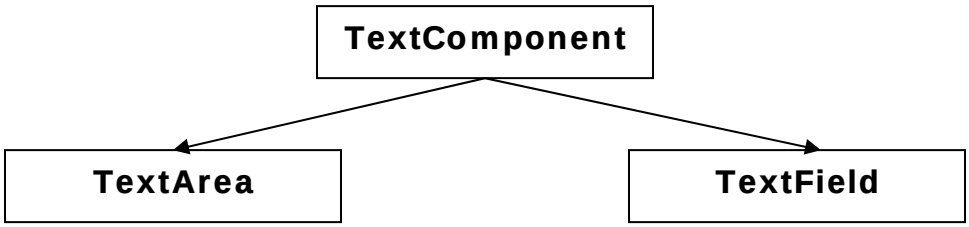


Fig. Text components hierarchy

The TextField class implements a single-line text-entry area, usually called an edit control. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections. TextField is a subclass of TextComponent. TextField defines the following constructors:

```
TextField( )
TextField(int numChars)
TextField(String str)
TextField(String str, int numChars)
```

The first version creates a default text field. The second form creates a text field that is *numChars* characters wide. The third form initializes the text field with the string contained in *str*. The fourth form initializes a text field and sets its width. TextField (and its superclass TextComponent) provides several methods that allow us to utilize a text field. In order to obtain the string currently contained in the text field, we can use getText( ). For setting the text, we call setText( ). These methods are as follows:

```
String getText( )
void setText(String str)
```

Here, *str* is the new string. The user can select a portion of the text in a text field. Also, we can select a portion of text under program control by using select( ). Our program can obtain the currently selected text by calling the getSelectedText( ). These methods are shown here:

```
String getSelectedText( )
void select(int startIndex, int endIndex)
```

The getSelectedText( ) returns the selected text. The select( ) method selects the characters beginning at *startIndex* and ending at *endIndex*–1. We can control whether the contents of a text field may be modified by the user by calling setEditable( ). We can determine editability by calling isEditable( ). These methods are shown here:

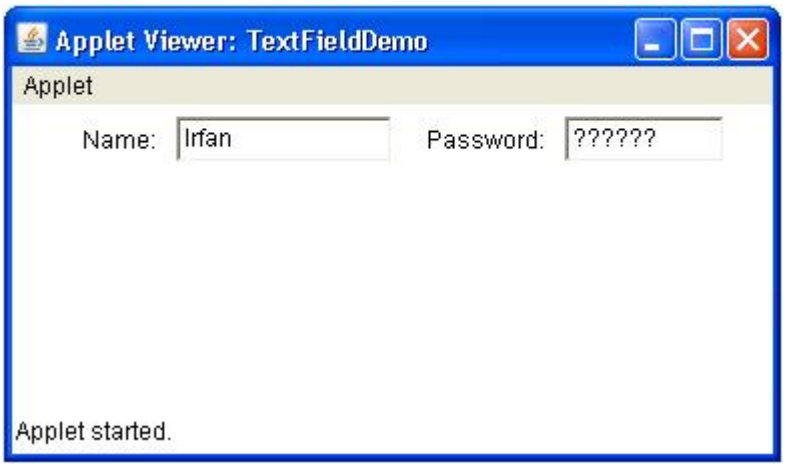
```
boolean isEditable( )
void setEditable(boolean canEdit)
```

The isEditable( ) returns true if the text may be changed and false if not. In setEditable( ), if canEdit is true, the text may be changed. If it is false, the text cannot be altered. There may be times when we will want the user to enter text that is not displayed, such as a password. You can disable the echoing of the characters as they are typed by calling setEchoChar( ). This method specifies a single character that the TextField will display when characters are entered (thus, the actual characters typed will not be shown). We can check a text field to see if it is in this mode with the echoCharIsSet( ) method. We can retrieve the echo character by calling the getEchoChar( ) method. These methods are as follows:

```
void setEchoChar(char ch)
boolean echoCharIsSet( )
char getEchoChar( )
```

Here, *ch* specifies the character to be echoed.

```
import java.awt.*;
import java.applet.*;
/*
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/
public class TextFieldDemo extends Applet
{
    TextField name, pass;
    public void init()
    {
        Label namep = new Label("Name: ", Label.RIGHT);
        Label passp = new Label("Password: ", Label.RIGHT);
        name = new TextField(12);
        pass = new TextField(8);
        pass.setEchoChar('?');
        add(namep);
        add(name);
        add(passp);
        add(pass);
    }
    public void paint(Graphics g)
    {
    }
}
```



TextArea

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called TextArea. Following are the constructors for TextArea:

```
TextArea( )
TextArea(int numLines, int numChars)
TextArea(String str)
TextArea(String str, int numLines, int numChars)
TextArea(String str, int numLines, int numChars, int sBars)
```

Here, *numLines* specifies the height, in lines, of the text area, and *numChars* specifies its width, in characters. Initial text can be specified by *str*. In the fifth form we can specify the scroll bars that we want the control to have. *sBars* must be one of these values:

SCROLLBARS_BOTH	SCROLLBARS_NONE
SCROLLBARS_HORIZONTAL_ONLY	SCROLLBARS_VERTICAL_ONLY

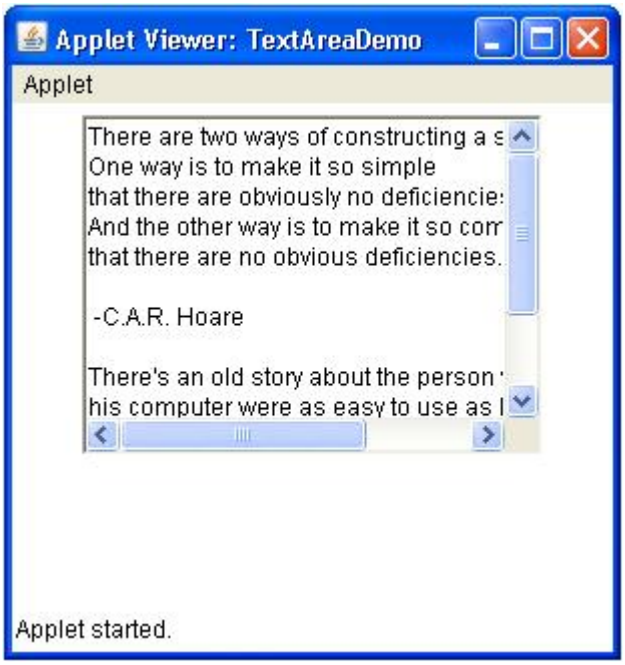
TextArea is a subclass of TextComponent. Therefore, it supports the getText( ), setText( ), getSelectedText( ), select( ), isEditable( ), and setEditable( ) methods described in the preceding section. TextArea adds the following methods:

```
void append(String str)
void insert(String str, int index)
void replaceRange(String str, int startIndex, int endIndex)
```

The append( ) method appends the string specified by *str* to the end of the current text. The insert( ) inserts the string passed in *str* at the specified index. In order to replace text, we call replaceRange( ). It replaces the characters from *startIndex* to *endIndex*–1, with the replacement text passed in *str*. Text areas are almost self-contained controls.

```
import java.awt.*;
import java.applet.*;
/*
<applet code="TextAreaDemo" width=300 height=250>
</applet>
*/
public class TextAreaDemo extends Applet
{
    public void init()
    {
        String val = "There are two ways of constructing " +
        "a software design.\n" +
        "One way is to make it so simple\n" +
        "that there are obviously no deficiencies.\n" +
        "And the other way is to make it so complicated\n" +
        "that there are no obvious deficiencies.\n\n" +
        " -C.A.R. Hoare\n\n" +
        "There's an old story about the person who wished\n" +
```

```
        "his computer were as easy to use as his telephone.\n" +
        "That wish has come true,\n" +
        "since I no longer know how to use my telephone.\n\n" +
        " -Bjarne Stroustrup, AT&T, (inventor of C++)";
        TextArea text = new TextArea(val, 10, 30);
        add(text);
    }
}
```



Layout Managers

All of the components that we have shown so far have been positioned by the default layout manager. A layout manager automatically arranges our controls within a window by using some type of algorithm. If we have programmed for other GUI environments, such as Windows, then we are accustomed to laying out our controls by hand. While it is possible to lay out Java controls by hand, too, we generally won't want to, for two main reasons. First, it is very tedious to manually lay out a large number of components. Second, sometimes the width and height information is not yet available when we need to arrange some control, because the native toolkit components haven't been realized. This is a chicken-and-egg situation; it is pretty confusing to figure out when it is okay to use the size of a given component to position it relative to another.

Each Container object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the setLayout( ) method. If no call to setLayout( ) is made, then the default layout manager is used. Whenever a

container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

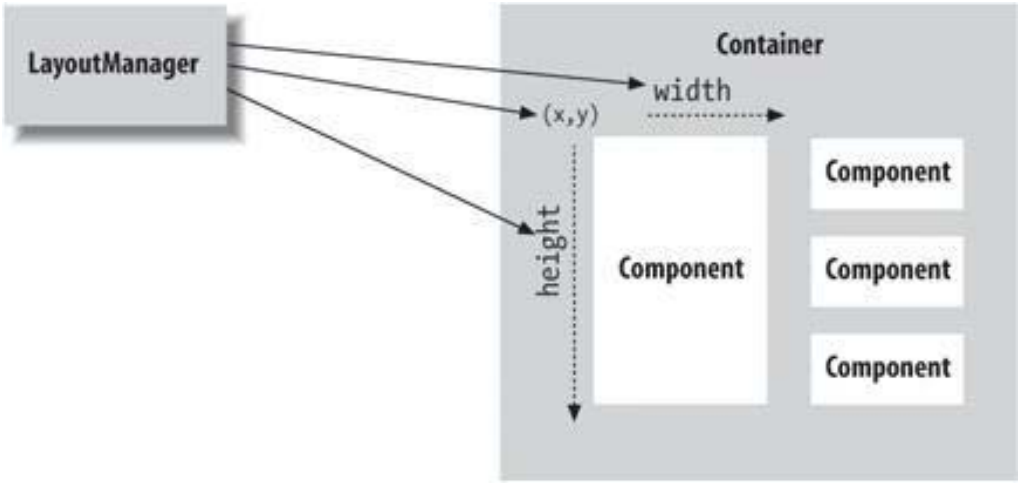


Fig. Layout Managers at work (Ref. No. 2)

The setLayout( ) method has the following general form:

```
void setLayout(LayoutManager layoutObj)
```

Here, *layoutObj* is a reference to the desired layout manager. If we wish to disable the layout manager and position components manually, pass null for *layoutObj*. If we do this, we will need to determine the shape and position of each component manually, using the setBounds( ) method defined by Component. Normally, we will want to use a layout manager.

Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time we add a component to a container. Whenever the container needs to be resized, the layout manager is consulted via its minimumLayoutSize( ) and preferredLayoutSize( ) methods. Each component that is being managed by a layout manager contains the getPreferredSize( ) and getMinimumSize( ) methods. These return the preferred and minimum size required to display each component. The layout manager will honor these requests if at all possible, while maintaining the integrity of the layout policy. We may override these methods for controls that we subclass. Default values are provided otherwise. Java has several predefined LayoutManager classes, several of which are described next. We can use the layout manager that best fits our application.

FlowLayout

FlowLayout is the default layout manager. This is the layout manager that the preceding examples have used. FlowLayout implements a simple layout style, which is similar to how words flow in a text editor. Components are laid out from the upper-left corner, left to right and top to bottom. When no more

components fit on a line, the next one appears on the next line. A small space is left between each component, above and below, as well as left and right. Here are the constructors for FlowLayout:

```
FlowLayout( )
FlowLayout(int how)
FlowLayout(int how, int horz, int vert)
```

The first form creates the default layout, which centers components and leaves five pixels of space between each component. The second form lets us specify how each line is aligned. Valid values for how are as follows:

```
FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT
```

These values specify left, center, and right alignment, respectively. The third form allows us to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Here is a version of the CheckboxDemo applet shown earlier, modified so that it uses left-aligned flow layout.

```
public class FlowLayoutDemo extends Applet
{
    String msg = "";
    Checkbox Win98, winNT, solaris, mac;
    public void init()
    {
        Win98 = new Checkbox("Windows 98/XP", null, true);
        winNT = new Checkbox("Windows NT/2000");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("MacOS");
        setLayout(new FlowLayout(FlowLayout.CENTER));
        add(Win98);
        add(winNT);
        add(solaris);
        add(mac);
    }
    public void paint(Graphics g)
    {
    }
}
```

BorderLayout

The BorderLayout class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center. Here are the constructors defined by BorderLayout:

```
BorderLayout( )
BorderLayout(int horz, int vert)
```

The first form creates a default border layout. The second allows us to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. BorderLayout defines the following constants that specify the regions:

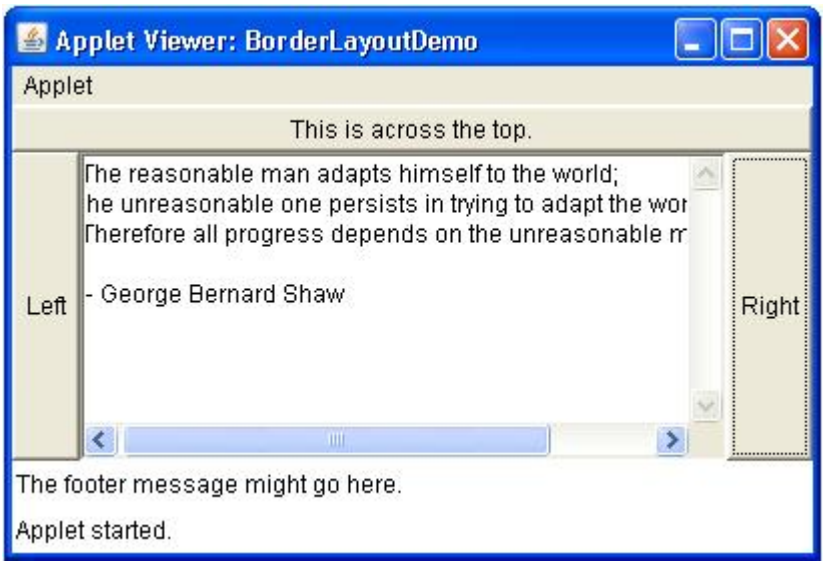
```
BorderLayout.CENTER      BorderLayout.SOUTH
BorderLayout.EAST        BorderLayout.WEST
BorderLayout.NORTH
```

When adding components, we will use these constants with the following form of add( ), which is defined by Container:

```
void add(Component compObj, Object region);
```

Here, *compObj* is the component to be added, and region specifies where the component will be added. Here is an example of a BorderLayout with a component in each layout area:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="BorderLayoutDemo" width=400 height=200>
</applet>
*/
public class BorderLayoutDemo extends Applet
{
    public void init()
    {
        setLayout(new BorderLayout());
        add(new Button("This is across the top."),
            BorderLayout.NORTH);
        add(new Label("The footer message might go here."),
            BorderLayout.SOUTH);
        add(new Button("Right"), BorderLayout.EAST);
        add(new Button("Left"), BorderLayout.WEST);
        String msg = "The reasonable man adapts " +
                     "himself to the world;\n" +
                     "the unreasonable one persists in " +
                     "trying to adapt the world to himself.\n" +
                     "Therefore all progress depends " +
                     "on the unreasonable man.\n\n" +
                     " - George Bernard Shaw\n\n";
        add(new TextArea(msg), BorderLayout.CENTER);
    }
}
```



Insets

Sometimes we will want to leave a small amount of space between the container that holds our components and the window that contains it. For doing this, we have to override the getInsets( ) method that is defined by Container. This function returns an Insets object that contains the top, bottom, left, and right inset to be used when the container is displayed. These values are used by the layout manager to inset the components when it lays out the window. The constructor for Insets is shown here:

```
Insets(int top, int left, int bottom, int right)
```

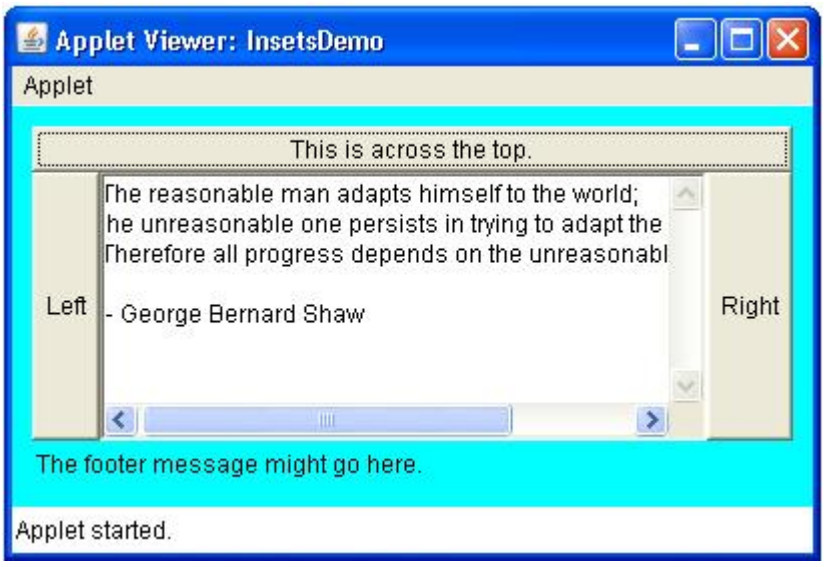
The values passed in top, left, bottom, and right specify the amount of space between the container and its enclosing window. The getInsets( ) method has this general form:

```
Insets getInsets( )
```

When overriding one of these methods, we must return a new Insets object that contains the inset spacing we desire. Here is the preceding BorderLayout example modified so that it insets its components ten pixels from each border. The background color has been set to cyan to help make the insets more visible.

```
public class InsetsDemo extends Applet
{
    public void init()
    {
        setBackground(Color.cyan);
        setLayout(new BorderLayout());
        add(new Button("This is across the top."),
                BorderLayout.NORTH);
        add(new Label("The footer message might go here."),
                BorderLayout.SOUTH);
        add(new Button("Right"), BorderLayout.EAST);
        add(new Button("Left"), BorderLayout.WEST);
        String msg = "The reasonable man adapts " +
                 "himself to the world;\n" +
                 "the unreasonable one persists in " +
                 "trying to adapt the world to himself.\n" +
                 "Therefore all progress depends " +
                 "on the unreasonable man.\n\n" +
                 " - George Bernard Shaw\n\n";
        add(new TextArea(msg), BorderLayout.CENTER);
    }
    public Insets getInsets()
    {
```

```
        return new Insets(10, 10, 10, 10);
    }
}
```



GridLayout

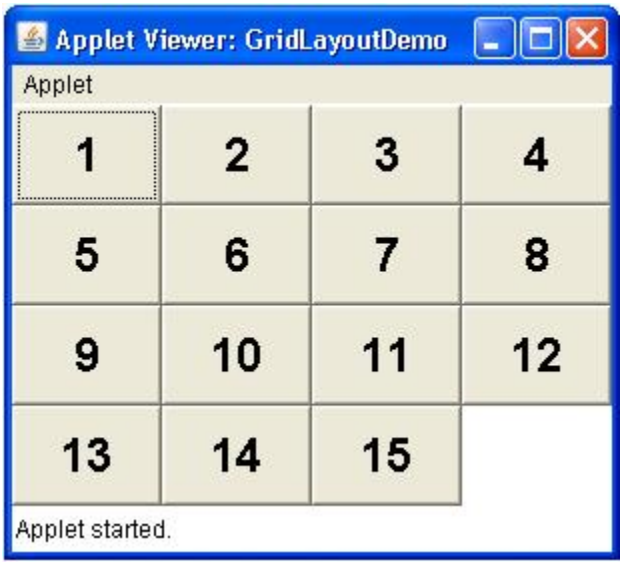
GridLayout lays out components in a two-dimensional grid. When we instantiate a GridLayout, we define the number of rows and columns. The constructors supported by GridLayout are shown here:

```
GridLayout( )
GridLayout(int numRows, int numColumns )
GridLayout(int numRows, int numColumns, int horz, int vert)
```

The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns. The third form allows us to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Either numRows or numColumns can be zero. Specifying *numRows* as zero allows for unlimited-length columns. Specifying *numColumns* as zero allows for unlimited-length rows. Here is a sample program that creates a 4×4 grid and fills it in with 15 buttons, each labeled with its index:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutDemo" width=300 height=200>
</applet>
*/
```

```
public class GridLayoutDemo extends Applet
{
    static final int n = 4;
    public void init()
    {
        setLayout(new GridLayout(n, n));
        setFont(new Font("SansSerif", Font.BOLD, 24));
        for(int i = 0; i < n; i++)
        {
            for(int j = 0; j < n; j++)
            {
                int k = i * n + j;
                if(k > 0)
                add(new Button("" + k));
            }
        }
    }
}
```



Menu Bars and Menus

A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This concept is implemented in Java by the following classes: **MenuBar**, **Menu**, and **MenuItem**. In general, a menu bar contains one or more Menu objects. Each Menu object contains a list of MenuItem objects. Each MenuItem object represents something that can be selected by the user. Since Menu is a subclass of MenuItem, a hierarchy of nested submenus can be created. It is also possible to include checkable menu items. These are menu options of type CheckboxMenuItem and will have a check mark next to them when they are selected. For creating a menu bar, we first create an instance of

MenuBar. This class only defines the default constructor. Next, create instances of Menu that will define the selections displayed on the bar.

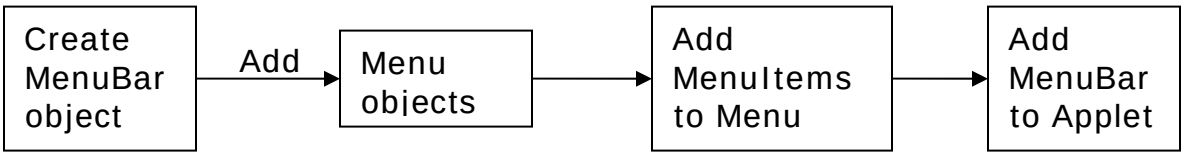


Fig. Creating a menu on Frame

Following are the constructors for Menu:

```
Menu( )
Menu(String optionName)
Menu(String optionName, boolean removable)
```

Here, *optionName* specifies the name of the menu selection. If removable is true, the pop-up menu can be removed and allowed to float free. Otherwise, it will remain attached to the menu bar. (Removable menus are implementation-dependent.) The first form creates an empty menu. Individual menu items are of type MenuItem. It defines these constructors:

```
MenuItem( )
MenuItem(String itemName)
MenuItem(String itemName, MenuShortcut keyAccel)
```

Here, *itemName* is the name shown in the menu, and *keyAccel* is the menu shortcut for this item. We can disable or enable a menu item by using the setEnabled( ) method. Its form is shown here:

```
void setEnabled(boolean enabledFlag)
```

If the argument *enabledFlag* is true, the menu item is enabled. If false, the menu item is disabled. We can determine an item's status by calling isEnabled( ). This method is shown here:

```
boolean isEnabled( )
```

The isEnabled( ) returns true if the menu item on which it is called is enabled. Otherwise, it returns false. We can change the name of a menu item by calling setLabel( ). We can retrieve the current name by using getLabel( ). These methods are as follows:

```
void setLabel(String newName)
String getLabel( )
```

Here, *newName* becomes the new name of the invoking menu item. getLabel( ) returns the current name. We can create a checkable menu item by using a subclass of MenuItem called CheckboxMenuItem. It has these constructors:

```
CheckboxMenuItem( )
CheckboxMenuItem(String itemName)
CheckboxMenuItem(String itemName, boolean on)
```

Here, *itemName* is the name shown in the menu. Checkable items operate as toggles. Each time one is selected, its state changes. In the first two forms, the checkable entry is unchecked. In the third form, if on is true, the checkable entry is initially checked. Otherwise, it is cleared. We can obtain the status of a checkable item by calling getState( ). We can set it to a known state by using setState( ). These methods are shown here:

```
boolean getState( )
void setState(boolean checked)
```

If the item is checked, getState( ) returns true. Otherwise, it returns false. For checking an item, pass true to setState( ). To clear an item, pass false. Once we have created a menu item, we must add the item to a Menu object by using add( ), which has the following general form:

```
MenuItem add(MenuItem item)
```

Here, *item* is the item being added. Items are added to a menu in the order in which the calls to add( ) take place. The item is returned. Once we have added all items to a Menu object, we can add that object to the menu bar by using this version of add( ) defined by MenuBar:

```
Menu add(Menu menu)
```

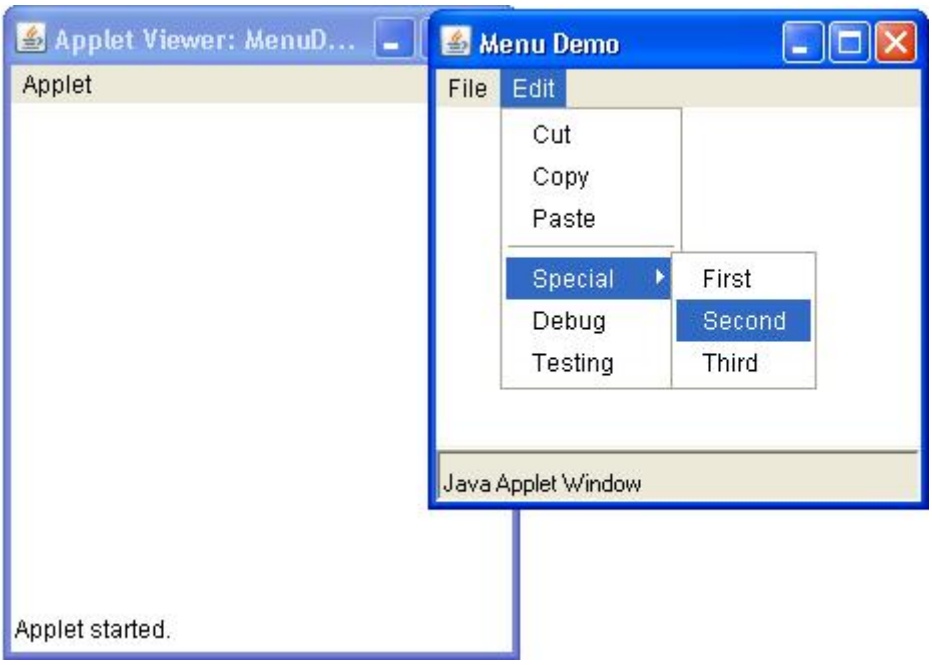
Here, menu is the menu being added. The menu is returned. Menus only generate events when an item of type MenuItem or CheckboxMenuItem is selected. They do not generate events when a menu bar is accessed to display a drop-down menu, for example. Each time a menu item is selected, an ActionEvent object is generated. Each time a check box menu item is checked or unchecked, an ItemEvent object is generated. Thus, we must implement the ActionListener and ItemListener interfaces in order to handle these menu events. The getItem( ) method of ItemEvent returns a reference to the item that generated this event. The general form of this method is shown here:

```
Object getItem( )
```

Following is an example that adds a series of nested menus to a pop-up window. The item selected is displayed in the window. The state of the two check box menu items is also displayed.

```
import java.awt.*;
import java.applet.*;
/*
<applet code="MenuDemo" width=250 height=250>
</applet>
*/
class MenuFrame extends Frame
{
    String msg = "";
    CheckboxMenuItem debug, test;
    MenuFrame(String title)
    {
        super(title);
        // create menu bar and add it to frame
        MenuBar mbar = new MenuBar();
        setMenuBar(mbar);
        // create the menu items
        Menu file = new Menu("File");
        MenuItem item1, item2, item3, item4, item5;
        file.add(item1 = new MenuItem("New..."));
        file.add(item2 = new MenuItem("Open..."));
        file.add(item3 = new MenuItem("Close"));
        file.add(item4 = new MenuItem("-"));
        file.add(item5 = new MenuItem("Quit..."));
        mbar.add(file);
        Menu edit = new Menu("Edit");
        MenuItem item6, item7, item8, item9;
        edit.add(item6 = new MenuItem("Cut"));
        edit.add(item7 = new MenuItem("Copy"));
        edit.add(item8 = new MenuItem("Paste"));
        edit.add(item9 = new MenuItem("-"));
        Menu sub = new Menu("Special");
        MenuItem item10, item11, item12;
        sub.add(item10 = new MenuItem("First"));
        sub.add(item11 = new MenuItem("Second"));
        sub.add(item12 = new MenuItem("Third"));
        edit.add(sub);
        // these are checkable menu items
        debug = new CheckboxMenuItem("Debug");
        edit.add(debug);
        test = new CheckboxMenuItem("Testing");
        edit.add(test);
        mbar.add(edit);
    }
```

```
}
public class MenuDemo extends Applet
{
    Frame f;
    public void init()
    {
        f = new MenuFrame("Menu Demo");
        f.setVisible(true);
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));
        setSize(width, height);
        f.setSize(width, height);
    }
    public void start()
    {
        f.setVisible(true);
    }
    public void stop()
    {
        f.setVisible(false);
    }
}
```



Dialog Boxes

Often, we will want to use a dialog box to hold a set of related controls. Dialog boxes are primarily used to obtain user input. They are similar to frame windows, except that dialog boxes are always child windows of a top-level

window. Also, dialog boxes don't have menu bars. In other respects, dialog boxes function like frame windows. (We can add controls to them, for example, in the same way that we add controls to a frame window.) Dialog boxes may be modal or modeless. When a modal dialog box is active, all input is directed to it until it is closed. This means that we cannot access other parts of our program until we have closed the dialog box. When a modeless dialog box is active, input focus can be directed to another window in our program. Thus, other parts of our program remain active and accessible. Dialog boxes are of type Dialog. Two commonly used constructors are shown here:

```
Dialog(Frame parentWindow, boolean mode)
Dialog(Frame parentWindow, String title, boolean mode)
```

Here, *parentWindow* is the owner of the dialog box. If mode is true, the dialog box is modal. Otherwise, it is modeless. The title of the dialog box can be passed in title. Generally, we will subclass Dialog, adding the functionality required by your application.

FileDialog

Java provides a built-in dialog box that lets the user specify a file. To create a file dialog box, instantiate an object of type *FileDialog*. This causes a file dialog box to be displayed. Usually, this is the standard file dialog box provided by the operating system. FileDialog provides these constructors:

```
FileDialog(Frame parent, String boxName)
FileDialog(Frame parent, String boxName, int how)
FileDialog(Frame parent)
```

Here, *parent* is the owner of the dialog box, and *boxName* is the name displayed in the box's title bar. If *boxName* is omitted, the title of the dialog box is empty. If *how* is FileDialog.LOAD, then the box is selecting a file for reading. If *how* is FileDialog.SAVE, the box is selecting a file for writing. The third constructor creates a dialog box for selecting a file for reading.

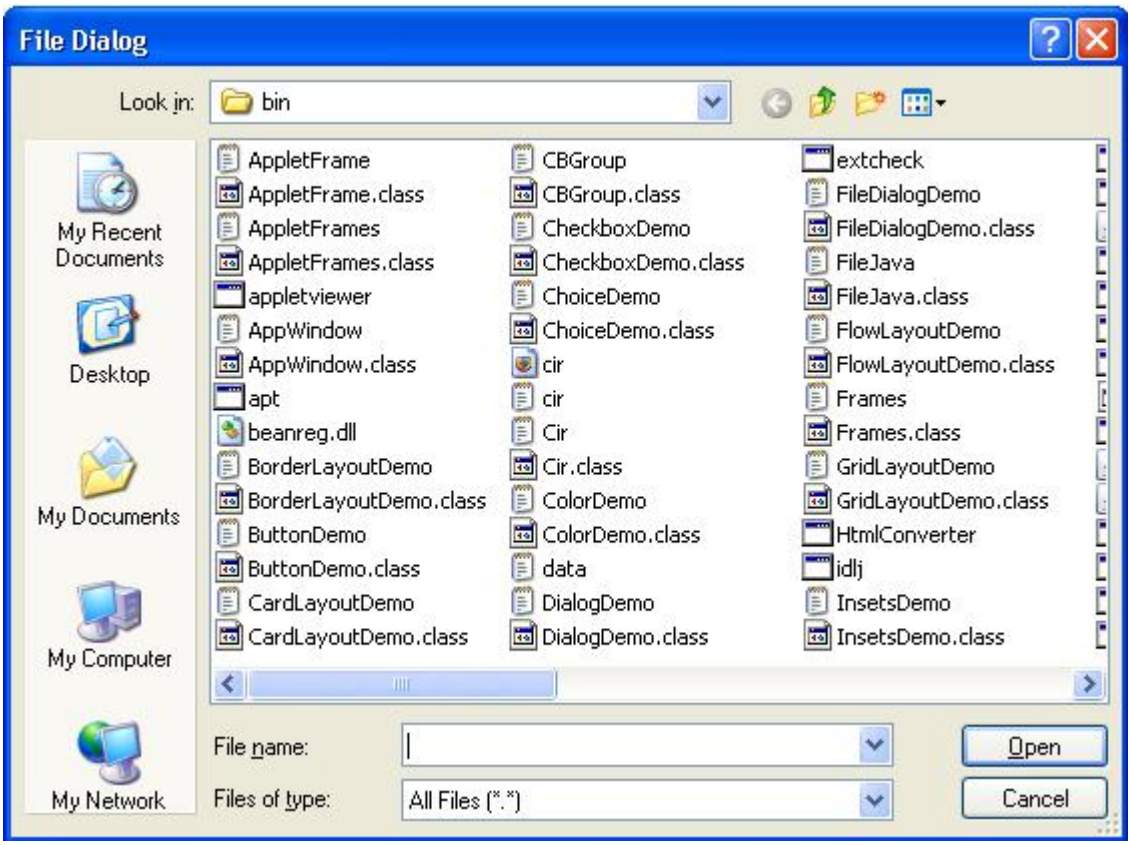
FileDialog( ) provides methods that allows us to determine the name of the file and its path as selected by the user. Here are two examples:

```
String getDirectory( )
String getFile( )
```

These methods return the directory and the filename, respectively. The following program activates the standard file dialog box:

```
import java.awt.*;
class SampleFrame extends Frame
{
```

```
    SampleFrame(String title)
    {
        super(title);
    }
}
class FileDialogDemo
{
    public static void main(String args[])
    {
        Frame f = new SampleFrame("File Dialog Demo");
        f.setVisible(true);
        f.setSize(100, 100);
        FileDialog fd = new FileDialog(f, "File Dialog");
        fd.setVisible(true);
    }
}
```



Event Handling

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

The Delegation Event Model

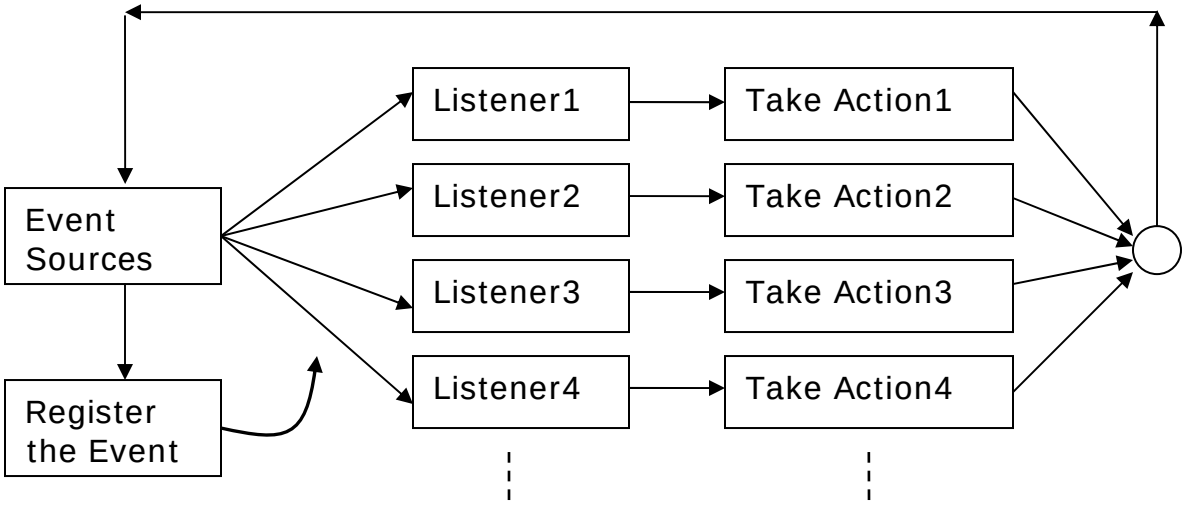


Fig. Delegation Event Model

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

In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. This is a more efficient way to handle events than the design used by the old Java 1.0 approach.

Java also allows us to process events without using the delegation event model. This can be done by extending an AWT component. However, the delegation event model is the preferred design for the reasons just cited.

Events

In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples. Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, software or hardware failure occurs, or an operation is completed. We are free to define events that are appropriate for your application.

Event Sources

A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

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

Here, *Type* is the name of the event and *el* is a reference to the event listener. For example, the method that registers a keyboard event listener is called addKeyListener( ). The method that registers a mouse motion listener is called addMouseMotionListener( ). When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as multicasting the event. In all cases, notifications are sent only to listeners that register to receive them. Some sources may allow only one listener to register. The general form of such a method is this:

```
public void addTypeListener(TypeListener el)
     throws java.util.TooManyListenersException
```

Here, *Type* is the name of the event and el is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as unicasting the event. A source must also provide a method that allows a listener to un-register an interest in a specific type of event. The general form of such a method is this:

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

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

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

Event Listeners

A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications. The methods that receive and process events are defined in a set of interfaces found in java.awt.event. For example, the MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface.

Event Classes

The classes that represent events are at the core of Java's event handling mechanism. At the root of the Java event class hierarchy is **EventObject**, which is in java.util. It is the superclass for all events. EventObject contains two methods: getSource( ) and toString( ). The getSource( ) method returns the source of the event. Its general form is shown here:

```
Object getSource( )
```

As expected, toString( ) returns the string equivalent of the event. The class AWTEvent, defined within the java.awt package, is a subclass of EventObject. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model.

- EventObject is a superclass of all events.
- AWTEvent is a superclass of all AWT events that are handled by the delegation event model.

The package java.awt.event defines several types of events that are generated by various user interface elements.

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.

FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract super class for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

ActionEvent

An *ActionEvent* is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. The ActionEvent class defines four integer constants that can be used to identify any modifiers associated with an action event: ALT_MASK, CTRL_MASK, META_MASK, and SHIFT_MASK. In addition, there is an integer constant, ACTION_PERFORMED, which can be used to identify action events. We can obtain the command name for the invoking ActionEvent object by using the getActionCommand( ) method, shown here:

```
String getActionCommand( )
```

For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button. The getModifiers( ) method returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. Its form is shown here:

```
int getModifiers( )
```

The method getWhen( ) that returns the time at which the event took place. This is called the event's timestamp. The getWhen( ) method is shown here.

```
long getWhen( )
```

AdjustmentEvent

An AdjustmentEvent is generated by a scroll bar. There are five types of adjustment events. The AdjustmentEvent class defines integer constants that

can be used to identify them. The constants and their meanings are shown here:

BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	The user clicked inside the scroll bar to increase its value.
TRACK	The slider was dragged.
UNIT_DECREMENT	The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	The button at the end of the scroll bar was clicked to increase its value.

The type of the adjustment event may be obtained by the getAdjustmentType( ) method. It returns one of the constants defined by AdjustmentEvent. The general form is shown here:

```
int getAdjustmentType( )
```

The amount of the adjustment can be obtained from the getValue( ) method, shown here:

```
int getValue( )
```

For example, when a scroll bar is manipulated, this method returns the value represented by that change.

ComponentEvent

A ComponentEvent is generated when the size, position, or visibility of a component is changed. There are four types of component events. The ComponentEvent class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

COMPONENT_HIDDEN	The component was hidden.
COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

ComponentEvent is the super-class either directly or indirectly of ContainerEvent, FocusEvent, KeyEvent, MouseEvent, and WindowEvent. The getComponent( ) method returns the component that generated the event. It is shown here:

```
Component getComponent( )
```

ContainerEvent

A ContainerEvent is generated when a component is added to or removed from a container. There are two types of container events. The ContainerEvent class defines int constants that can be used to identify them: COMPONENT_ADDED and COMPONENT_REMOVED. They indicate that a component has been added to or removed from the container.

FocusEvent

A FocusEvent is generated when a component gains or loses input focus. These events are identified by the integer constants FOCUS_GAINED and FOCUS_LOST. FocusEvent is a subclass of ComponentEvent.

If the user moves the mouse to adjust a scroll bar, the focus is temporarily lost.) The other component involved in the focus change, called the opposite component, is passed in other. Therefore, if a FOCUS_GAINED event occurred, other will refer to the component that lost focus. Conversely, if a FOCUS_LOST event occurred, other will refer to the component that gains focus.

InputEvent

The abstract class InputEvent is a subclass of ComponentEvent and is the superclass for component input events. Its subclasses are KeyEvent and MouseEvent. InputEvent defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event. Originally, the InputEvent class defined the following eight values to represent the modifiers.

ALT_MASK	BUTTON2_MASK META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK SHIFT_MASK
	BUTTON1_MASK CTRL_MASK

However, because of possible conflicts between the modifiers used by keyboard events and mouse events, and other issues, Java 2, version 1.4 added the following extended modifier values.

ALT_DOWN_MASK	ALT_GRAPH_DOWN_MASK
BUTTON1_DOWN_MASK	BUTTON2_DOWN_MASK
BUTTON3_DOWN_MASK	CTRL_DOWN_MASK
META_DOWN_MASK	SHIFT_DOWN_MASK

When writing new code, it is recommended that we use the new, extended modifiers rather than the original modifiers. To test if a modifier was pressed at the time an event is generated, use the isAltDown( ),

isAltGraphDown( ), isControlDown( ), isMetaDown( ), and isShiftDown( ) methods. The forms of these methods are shown here:

```
boolean isAltDown( )
boolean isAltGraphDown( )
boolean isControlDown( )
boolean isMetaDown( )
boolean isShiftDown( )
```

We can obtain a value that contains all of the original modifier flags by calling the getModifiers( ) method. It is shown here:

```
int getModifiers( )
```

We can obtain the extended modifiers by called getModifiersEx( ), which is shown here.

```
int getModifiersEx( )
```

ItemEvent

An ItemEvent is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected. There are two types of item events, which are identified by the following integer constants:

DESELECTED	The user deselected an item.
SELECTED	The user selected an item.

In addition, ItemEvent defines one integer constant, ITEM_STATE_CHANGED, that signifies a change of state.

The getItem( ) method can be used to obtain a reference to the item that generated an event. Its signature is shown here:

```
Object getItem( )
```

The getItemSelectable( ) method can be used to obtain a reference to the ItemSelectable object that generated an event. Its general form is shown here:

```
ItemSelectable getItemSelectable( )
```

Lists and choices are examples of user interface elements that implement the ItemSelectable interface. The getStateChange( ) method returns the state change (i.e., SELECTED or DESELECTED) for the event. It is shown here:

```
int getStateChange( )
```

KeyEvent

A KeyEvent is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: KEY_PRESSED, KEY_RELEASED, and KEY_TYPED. The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember, not all key presses result in characters. For example, pressing the SHIFT key does not generate a character. There are many other integer constants that are defined by KeyEvent. For example, VK_0 through VK_9 and VK_A through VK_Z define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ENTER	VK_ESCAPE	VK_CANCEL	VK_UP
VK_DOWN	VK_LEFT	VK_RIGHT	VK_PAGE_DOWN
VK_PAGE_UP	VK_SHIFT	VK_ALT	VK_CONTROL

The VK constants specify virtual key codes and are independent of any modifiers, such as control, shift, or alt. KeyEvent is a subclass of InputEvent.

The KeyEvent class defines several methods, but the most commonly used ones are getKeyChar( ), which returns the character that was entered, and getKeyCode( ), which returns the key code. Their general forms are shown here:

```
char getKeyChar( )
int getKeyCode( )
```

If no valid character is available, then getKeyChar( ) returns CHAR_UNDEFINED. When a KEY_TYPED event occurs, getKeyCode( ) returns VK_UNDEFINED.

MouseEvent

There are eight types of mouse events. The MouseEvent class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED	The user clicked the mouse.
MOUSE_DRAGGED	The user dragged the mouse.
MOUSE_ENTERED	The mouse entered a component.
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved.
MOUSE_PRESSED	The mouse was pressed.
MOUSE_RELEASED	The mouse was released.
MOUSE_WHEEL	The mouse wheel was moved.

The most commonly used methods in this class are getX( ) and getY( ). These returns the X and Y coordinate of the mouse when the event occurred. Their forms are shown here:

```
int getX( )
int getY( )
```

Alternatively, we can use the getPoint( ) method to obtain the coordinates of the mouse. It is shown here:

```
Point getPoint( )
```

It returns a Point object that contains the X, Y coordinates in its integer members: x and y. The translatePoint( ) method changes the location of the event. Its form is shown here:

```
void translatePoint(int x, int y)
```

Here, the arguments x and y are added to the coordinates of the event. The getClickCount( ) method obtains the number of mouse clicks for this event. Its signature is shown here:

```
int getClickCount( )
```

The isPopupTrigger( ) method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:

```
boolean isPopupTrigger( )
```

Java 2, version 1.4 added the getButton( ) method, shown here.

```
int getButton( )
```

It returns a value that represents the button that caused the event. The return value will be one of these constants defined by MouseEvent.

NOBUTTON BUTTON1 BUTTON2 BUTTON3

The NOBUTTON value indicates that no button was pressed or released.

MouseWheelEvent

The MouseWheelEvent class encapsulates a mouse wheel event. It is a subclass of MouseEvent and was added by Java 2, version 1.4. Not all mice have wheels. If a mouse has a wheel, it is located between the left and right

buttons. Mouse wheels are used for scrolling. MouseWheelEvent defines these two integer constants.

WHEEL_BLOCK_SCROLL	A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL	A line-up or line-down scroll event occurred.

MouseWheelEvent defines methods that give us access to the wheel event. For obtaining the number of rotational units, call getWheelRotation( ), shown here.

```
int getWheelRotation( )
```

It returns the number of rotational units. If the value is positive, the wheel moved counterclockwise. If the value is negative, the wheel moved clockwise. For obtaining the type of scroll, call getScrollType( ), shown next.

```
int getScrollType( )
```

It returns either WHEEL_UNIT_SCROLL or WHEEL_BLOCK_SCROLL. If the scroll type is WHEEL_UNIT_SCROLL, we can obtain the number of units to scroll by calling getScrollAmount( ). It is shown here.

```
int getScrollAmount( )
```

TextEvent

Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program. TextEvent defines the integer constant TEXT_VALUE_CHANGED.

The TextEvent object does not include the characters currently in the text component that generated the event. Instead, our program must use other methods associated with the text component to retrieve that information. This operation differs from other event objects discussed in this section. For this reason, no methods are discussed here for the TextEvent class. Think of a text event notification as a signal to a listener that it should retrieve information from a specific text component.

WindowEvent

There are ten types of window events. The WindowEvent class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.

WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window gained input focus.
WINDOW_ICONIFIED	The window was iconified.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.
WINDOW_STATE_CHANGED	The state of the window changed.

WindowEvent is a subclass of ComponentEvent. The most commonly used method in this class is getWindow( ). It returns the Window object that generated the event. Its general form is shown here:

```
Window getWindow( )
```

Java 2, version 1.4, adds methods that return the opposite window (when a focus event has occurred), the previous window state, and the current window state. These methods are shown here:

```
Window getOppositeWindow()
int getOldState()
int getNewState()
```

Sources of Events

Following is list of some of the user interface components that can generate the events described in the previous section. In addition to these graphical user interface elements, other components, such as an applet, can generate events. For example, we receive key and mouse events from an applet. (We may also build our own components that generate events.)

Event Source	*Description*
Button	Generates action events when the button is pressed.
Checkbox	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; Generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scrollbar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.

Window	Generates window events when a window is activated, closed,deactivated, deiconified, iconified, opened, or quit.

Event Listener Interfaces

The delegation event model has two parts: sources and listeners. Listeners are created by implementing one or more of the interfaces defined by the java.awt.event package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument.

ActionListener Interface

This interface defines the actionPerformed( ) method that is invoked when an action event occurs. Its general form is shown here:

```
void actionPerformed(ActionEvent ae)
```

AdjustmentListener Interface

This interface defines the adjustmentValueChanged( ) method that is invoked when an adjustment event occurs. Its general form is shown here:

```
void adjustmentValueChanged(AdjustmentEvent ae)
```

ComponentListener Interface

This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:

```
void componentResized(ComponentEvent ce)
void componentMoved(ComponentEvent ce)
void componentShown(ComponentEvent ce)
void componentHidden(ComponentEvent ce)
```

The AWT processes the resize and move events. The componentResized() and componentMoved( ) methods are provided for notification purposes only.

ContainerListener Interface

This interface contains two methods. When a component is added to a container, componentAdded( ) is invoked. When a component is removed from a container, componentRemoved( ) is invoked. Their general forms are shown here:

```
void componentAdded(ContainerEvent ce)
void componentRemoved(ContainerEvent ce)
```

FocusListener Interface

This interface defines two methods. When a component obtains keyboard focus, focusGained( ) is invoked. When a component loses keyboard focus, focusLost( ) is called. Their general forms are shown here:

```
void focusGained(FocusEvent fe)
void focusLost(FocusEvent fe)
```

ItemListener Interface

This interface defines the itemStateChanged( ) method that is invoked when the state of an item changes. Its general form is shown here:

```
void itemStateChanged(ItemEvent ie)
```

KeyListener Interface

This interface defines three methods. The keyPressed( ) and keyReleased( ) methods are invoked when a key is pressed and released, respectively. The keyTyped( ) method is invoked when a character has been entered. For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the HOME key, two key events are generated in sequence: key pressed and released. The general forms of these methods are shown here:

```
void keyPressed(KeyEvent ke)
void keyReleased(KeyEvent ke)
void keyTyped(KeyEvent ke)
```

MouseListener Interface

This interface defines five methods. If the mouse is pressed and released at the same point, mouseClicked( ) is invoked. When the mouse enters a component, the mouseEntered( ) method is called. When it leaves, mouseExited( ) is called. The mousePressed( ) and mouseReleased( ) methods are invoked when the mouse is pressed and released, respectively. The general forms of these methods are shown here:

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
```

```
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
```

MouseMotionListener Interface

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

```
void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)
```

MouseWheelListener Interface

This interface defines the mouseWheelMoved( ) method that is invoked when the mouse wheel is moved. Its general form is shown here.

```
void mouseWheelMoved(MouseWheelEvent mwe)
```

MouseWheelListener was added by Java 2, version 1.4.

TextListener Interface

This interface defines the textChanged( ) method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

```
void textChanged(TextEvent te)
```

WindowFocusListener Interface

This interface defines two methods: windowGainedFocus( ) and windowLostFocus( ). These are called when a window gains or losses input focus. Their general forms are shown here.

```
void windowGainedFocus(WindowEvent we)
void windowLostFocus(WindowEvent we)
```

WindowFocusListener was added by Java 2, version 1.4.

WindowListener Interface

This interface defines seven methods. The windowActivated( ) and windowDeactivated( ) methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the windowIconified( ) method is called. When a window is deiconified, the windowDeiconified( )

method is called. When a window is opened or closed, the windowOpened( ) or windowClosed( ) methods are called, respectively. The windowClosing( ) method is called when a window is being closed. The general forms of these methods are:

```
void windowActivated(WindowEvent we)
void windowClosed(WindowEvent we)
void windowClosing(WindowEvent we)
void windowDeactivated(WindowEvent we)
void windowDeiconified(WindowEvent we)
void windowIconified(WindowEvent we)
void windowOpened(WindowEvent we)
```

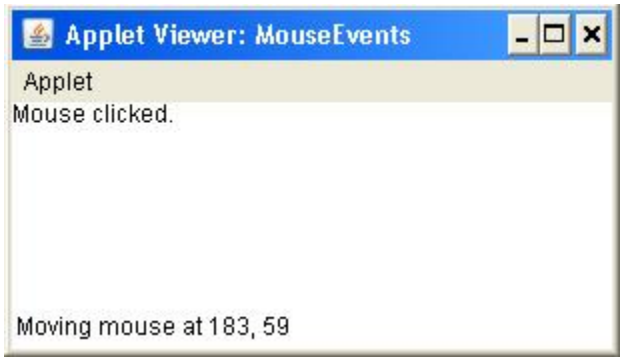
Handling Mouse Events

In order to handle mouse events, we must implement the MouseListener and the MouseMotionListener interfaces.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</applet>
*/
public class MouseEvents extends Applet
implements MouseListener, MouseMotionListener
{
    String msg = "";
    int mouseX = 0, mouseY = 0; // coordinates of mouse
    public void init()
    {
        addMouseListener(this);
        addMouseMotionListener(this);
    }
    public void mouseClicked(MouseEvent me)
    {
        // save coordinates
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse clicked.";
        repaint();
    }
    // Handle mouse entered.
    public void mouseEntered(MouseEvent me)
    {
        // save coordinates
        mouseX = 0;
```

```
        mouseY = 10;
        msg = "Mouse entered.";
        repaint();
    }
    // Handle mouse exited.
    public void mouseExited(MouseEvent me)
    {
        // save coordinates
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse exited.";
        repaint();
    }
    // Handle button pressed.
    public void mousePressed(MouseEvent me)
    {
        // save coordinates
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "Down";
        repaint();
    }
    // Handle button released.
    public void mouseReleased(MouseEvent me)
    {
        // save coordinates
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "Up";
        repaint();
    }
    // Handle mouse dragged.
    public void mouseDragged(MouseEvent me)
    {
        // save coordinates
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "*";
        showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
        repaint();
    }
    // Handle mouse moved.
    public void mouseMoved(MouseEvent me)
    {
        // show status
        showStatus("Moving mouse at " + me.getX() + ", " +
          me.getY());
    }
```

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



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

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

```
void addMouseListener(MouseListener ml)
void addMouseMotionListener(MouseMotionListener mml)
```

Here, *ml* is a reference to the object receiving mouse events, and *mml* is a reference to the object receiving mouse motion events. In this program, the same object is used for both. The applet then implements all of the methods defined by the MouseListener and MouseMotionListener interfaces. These are the event handlers for the various mouse events. Each method handles its event and then returns.

Handling Keyboard Events

We will be implementing the KeyListener interface for handling keyboard events. Before looking at an example, it is useful to review how key events are generated. When a key is pressed, a KEY_PRESSED event is generated. This results in a call to the keyPressed( ) event handler. When the key is released, a KEY_RELEASED event is generated and the keyReleased( ) handler is executed. If a character is generated by the keystroke, then a KEY_TYPED event is sent

and the keyTyped( ) handler is invoked. Thus, each time the user presses a key, at least two and often three events are generated. If all we care about are actual characters, then we can ignore the information passed by the key press and release events. However, if our program needs to handle special keys, such as the arrow or function keys, then it must watch for them through the keyPressed( ) handler.

There is one other requirement that our program must meet before it can process keyboard events: it must request input focus. To do this, call requestFocus( ), which is defined by Component. If we don't, then our program will not receive any keyboard events. The following program demonstrates keyboard input. It echoes keystrokes to the applet window and shows the pressed/released status of each key in the status window.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SimpleKey" width=300 height=100>
</applet>
*/
public class SimpleKey extends Applet
implements KeyListener
{
    String msg = "";
    int X = 10, Y = 20; // output coordinates
    public void init()
    {
        addKeyListener(this);
        requestFocus(); // request input focus
    }
    public void keyPressed(KeyEvent ke)
    {
        showStatus("Key Down");
    }
    public void keyReleased(KeyEvent ke)
    {
        showStatus("Key Up");
    }
    public void keyTyped(KeyEvent ke)
    {
        msg += ke.getKeyChar();
        repaint();
    }
    // Display keystrokes.
    public void paint(Graphics g)
    {
        g.drawString(msg, X, Y);
```

```
    }
}
```

If we want to handle the special keys, such as the arrow or function keys, we need to respond to them within the keyPressed( ) handler. They are not available through keyTyped( ). To identify the keys, we use their virtual key codes. For example, the next method shows the use of special keys:

```
public void keyPressed(KeyEvent ke)
{
     showStatus("Key Down");
     int key = ke.getKeyCode();
     switch(key)
     {
          case KeyEvent.VK_F1:
               msg += "<F1>";
               break;
          case KeyEvent.VK_F2:
               msg += "<F2>";
               break;
          case KeyEvent.VK_F3:
               msg += "<F3>";
               break;
          case KeyEvent.VK_PAGE_DOWN:
               msg += "<PgDn>";
               break;
          case KeyEvent.VK_PAGE_UP:
               msg += "<PgUp>";
               break;
          case KeyEvent.VK_LEFT:
               msg += "<Left Arrow>";
               break;
          case KeyEvent.VK_RIGHT:
               msg += "<Right Arrow>";
               break;
     }
     repaint();
```

```
}
```

Adapter Classes

Java provides a special feature, called an adapter class that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when we want to receive and process only some of the events that are handled by a particular event listener interface. We can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which we are interested. For example, the MouseMotionAdapter class has two methods, mouseDragged( ) and mouseMoved( ). The signatures of these empty methods are exactly as defined in the MouseMotionListener interface. If you were interested in only mouse drag events, then you could simply extend MouseMotionAdapter and implement mouseDragged( ). The empty implementation of mouseMoved( ) would handle the mouse motion events for you.

List below shows the commonly used adapter classes in java.awt.event and notes the interface that each implements. The following example demonstrates an adapter. It displays a message in the status bar of an applet viewer or browser when the mouse is clicked or dragged. However, all other mouse events are silently ignored. The program has three classes. AdapterDemo extends Applet. Its init( ) method creates an instance of MyMouseAdapter and registers that object to receive notifications of mouse events. It also creates an instance of MyMouseMotionAdapter and registers that object to receive notifications of mouse motion events. Both of the constructors take a reference to the applet as an argument. MyMouseAdapter implements the mouseClicked( ) method. The other mouse events are silently ignored by code inherited from the MouseAdapter class. MyMouseMotionAdapter implements the mouseDragged( ) method. The other mouse motion event is silently ignored by code inherited from the MouseMotionAdapter class.

Adapter Class	**Listener Interface**
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

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

```
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/
public class AdapterDemo extends Applet
{
    public void init()
    {
        addMouseListener(new MyMouseAdapter(this));
        addMouseMotionListener(new MyMouseMotionAdapter(this));
    }
}
class MyMouseAdapter extends MouseAdapter
{
    AdapterDemo adapterDemo;
    public MyMouseAdapter(AdapterDemo adapterDemo)
    {
        this.adapterDemo = adapterDemo;
    }
    // Handle mouse clicked.
    public void mouseClicked(MouseEvent me)
    {
        adapterDemo.showStatus("Mouse clicked");
    }
}
class MyMouseMotionAdapter extends MouseMotionAdapter
{
    AdapterDemo adapterDemo;
    public MyMouseMotionAdapter(AdapterDemo adapterDemo)
    {
        this.adapterDemo = adapterDemo;
    }
    // Handle mouse dragged.
    public void mouseDragged(MouseEvent me)
    {
        adapterDemo.showStatus("Mouse dragged");
    }
}
```

As we can see by looking at the program, not having to implement all of the methods defined by the MouseMotionListener and MouseListener interfaces saves our considerable amount of effort and prevents our code from becoming cluttered with empty methods.

Inner Classes

For understanding the benefit provided by inner classes, consider the applet shown in the following listing. It does not use an inner class. Its goal is to display the string “Mouse Pressed” in the status bar of the applet viewer or browser when the mouse is pressed. There are two top-level classes in this program. MousePressedDemo extends Applet, and MyMouseAdapter extends MouseAdapter. The init( ) method of MousePressedDemo instantiates MyMouseAdapter and provides this object as an argument to the addMouseListener( ) method. Notice that a reference to the applet is supplied as an argument to the MyMouseAdapter constructor. This reference is stored in an instance variable for later use by the mousePressed( ) method. When the mouse is pressed, it invokes the showStatus( ) method of the applet through the stored applet reference. In other words, showStatus( ) is invoked relative to the applet reference stored by MyMouseAdapter.

```
// This applet does NOT use an inner class.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="MousePressedDemo" width=200 height=100>
</applet>
*/
public class MousePressedDemo extends Applet
{
     public void init()
     {
          addMouseListener(new MyMouseAdapter(this));
     }
}
class MyMouseAdapter extends MouseAdapter
{
     MousePressedDemo mousePressedDemo;
     public MyMouseAdapter(MousePressedDemo mousePressedDemo)
     {
          this.mousePressedDemo = mousePressedDemo;
     }
     public void mousePressed(MouseEvent me)
     {
          mousePressedDemo.showStatus("Mouse Pressed.");
     }
}
```

The following listing shows how the preceding program can be improved by using an inner class. Here, InnerClassDemo is a top-level class that extends Applet. MyMouseAdapter is an inner class that extends MouseAdapter. Because MyMouseAdapter is defined within the scope of InnerClassDemo, it has access

to all of the variables and methods within the scope of that class. Therefore, the mousePressed( ) method can call the showStatus( ) method directly. It no longer needs to do this via a stored reference to the applet. Thus, it is no longer necessary to pass MyMouseAdapter( ) a reference to the invoking object.

```
// Inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="InnerClassDemo" width=200 height=100>
</applet>
*/
public class InnerClassDemo extends Applet
{
     public void init()
     {
          addMouseListener(new MyMouseAdapter());
     }
     class MyMouseAdapter extends MouseAdapter
     {
          public void mousePressed(MouseEvent me)
          {
               showStatus("Mouse Pressed");
          }
     }
}
```

Anonymous Inner Classes

An anonymous inner class is one that is not assigned a name. Consider the applet shown in the following listing. As before, its goal is to display the string “Mouse Pressed” in the status bar of the applet viewer or browser when the mouse is pressed.

```
// Anonymous inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="AnonymousInnerClassDemo" width=200 height=100>
</applet>
*/
public class AnonymousInnerClassDemo extends Applet
{
     public void init()
     {
          addMouseListener(new MouseAdapter() {
               public void mousePressed(MouseEvent me) {
```

```
                showStatus("Mouse Pressed");
                }
          });
     }
}
```

There is one top-level class in this program: AnonymousInnerClassDemo. The init( ) method calls the addMouseListener( ) method. Its argument is an expression that defines and instantiates an anonymous inner class. Let's analyze this expression carefully. The syntax new MouseAdapter( ) { ... } indicates to the compiler that the code between the braces defines an anonymous inner class. Furthermore, that class extends MouseAdapter. This new class is not named, but it is automatically instantiated when this expression is executed. Because this anonymous inner class is defined within the scope of AnonymousInnerClassDemo, it has access to all of the variables and methods within the scope of that class. Therefore, it can call the showStatus( ) method directly. As just illustrated, both named and anonymous inner classes solve some annoying problems in a simple yet effective way. They also allow us to create more efficient code.

Handling Buttons

```
// Demonstrate Buttons
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonDemo" width=250 height=150>
</applet>
*/
public class ButtonDemo extends Applet implements ActionListener
{
     String msg = "";
     Button yes, no, maybe;
     public void init()
     {
          yes = new Button("Yes");
          no = new Button("No");
          maybe = new Button("Undecided");
          add(yes);
          add(no);
          add(maybe);
          yes.addActionListener(this);
          no.addActionListener(this);
          maybe.addActionListener(this);
     }
     public void actionPerformed(ActionEvent ae)
```

```
     {
          String str = ae.getActionCommand();
          if(str.equals("Yes"))
          {
               msg = "You pressed Yes.";
          }
          else if(str.equals("No"))
          {
               msg = "You pressed No.";
          }
               else
               {
                    msg = "You pressed Undecided.";
               }
          repaint();
     }
     public void paint(Graphics g)
     {
          g.drawString(msg, 6, 100);
     }
}
```

Handling Checkboxes

```
// Demonstrate check boxes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CheckboxDemo" width=250 height=200>
</applet>
*/
public class CheckboxDemo extends Applet implements ItemListener
{
     String msg = "";
     Checkbox Win98, winNT, solaris, mac;
     public void init()
     {
          Win98 = new Checkbox("Windows 98/XP", null, true);
          winNT = new Checkbox("Windows NT/2000");
          solaris = new Checkbox("Solaris");
          mac = new Checkbox("MacOS");
          add(Win98);
          add(winNT);
          add(solaris);
          add(mac);
          Win98.addItemListener(this);
          winNT.addItemListener(this);
```

```
          solaris.addItemListener(this);
          mac.addItemListener(this);
     }
     public void itemStateChanged(ItemEvent ie)
     {
          repaint();
     }
     // Display current state of the check boxes.
     public void paint(Graphics g)
     {
          msg = "Current state: ";
          g.drawString(msg, 6, 80);
          msg = " Windows 98/XP: " + Win98.getState();
          g.drawString(msg, 6, 100);
          msg = " Windows NT/2000: " + winNT.getState();
          g.drawString(msg, 6, 120);
          msg = " Solaris: " + solaris.getState();
          g.drawString(msg, 6, 140);
          msg = " MacOS: " + mac.getState();
          g.drawString(msg, 6, 160);
     }
}
```

Handling Radio Buttons

```
// Demonstrate check box group.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CBGroup" width=250 height=200>
</applet>
*/
public class CBGroup extends Applet implements ItemListener
{
     String msg = "";
     Checkbox Win98, winNT, solaris, mac;
     CheckboxGroup cbg;
     public void init()
     {
          cbg = new CheckboxGroup();
          Win98 = new Checkbox("Windows 98/XP", cbg, true);
          winNT = new Checkbox("Windows NT/2000", cbg, false);
          solaris = new Checkbox("Solaris", cbg, false);
          mac = new Checkbox("MacOS", cbg, false);
          add(Win98);
          add(winNT);
          add(solaris);
```

```
          add(mac);
          Win98.addItemListener(this);
          winNT.addItemListener(this);
          solaris.addItemListener(this);
          mac.addItemListener(this);
     }
     public void itemStateChanged(ItemEvent ie)
     {
          repaint();
     }
     // Display current state of the check boxes.
     public void paint(Graphics g)
     {
          msg = "Current selection: ";
          msg += cbg.getSelectedCheckbox().getLabel();
          g.drawString(msg, 6, 100);
     }
}
```

Handling Choice Controls

```
// Demonstrate Choice lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ChoiceDemo" width=300 height=180>
</applet>
*/
public class ChoiceDemo extends Applet implements ItemListener
{
     Choice os, browser;
     String msg = "";
     public void init()
     {
          os = new Choice();
          browser = new Choice();
          // add items to os list
          os.add("Windows 98/XP");
          os.add("Windows NT/2000");
          os.add("Solaris");
          os.add("MacOS");
          // add items to browser list
          browser.add("Netscape 3.x");
          browser.add("Netscape 4.x");
          browser.add("Netscape 5.x");
          browser.add("Netscape 6.x");
          browser.add("Internet Explorer 4.0");
```

```
          browser.add("Internet Explorer 5.0");
          browser.add("Internet Explorer 6.0");
          browser.add("Lynx 2.4");
          browser.select("Netscape 4.x");
          // add choice lists to window
          add(os);
          add(browser);
          // register to receive item events
          os.addItemListener(this);
          browser.addItemListener(this);
     }
     public void itemStateChanged(ItemEvent ie)
     {
          repaint();
     }
     // Display current selections.
     public void paint(Graphics g)
     {
          msg = "Current OS: ";
          msg += os.getSelectedItem();
          g.drawString(msg, 6, 120);
          msg = "Current Browser: ";
          msg += browser.getSelectedItem();
          g.drawString(msg, 6, 140);
     }
}
```

Handling Lists

```
// Demonstrate Lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ListDemo" width=300 height=180>
</applet>
*/
public class ListDemo extends Applet implements ActionListener
{
     List os, browser;
     String msg = "";
     public void init()
     {
          os = new List(4, true);
          browser = new List(4, false);
          // add items to os list
          os.add("Windows 98/XP");
          os.add("Windows NT/2000");
```

```
          os.add("Solaris");
          os.add("MacOS");
          // add items to browser list
          browser.add("Netscape 3.x");
          browser.add("Netscape 4.x");
          browser.add("Netscape 5.x");
          browser.add("Netscape 6.x");
          browser.add("Internet Explorer 4.0");
          browser.add("Internet Explorer 5.0");
          browser.add("Internet Explorer 6.0");
          browser.add("Lynx 2.4");
          browser.select(1);
          // add lists to window
          add(os);
          add(browser);
          // register to receive action events
          os.addActionListener(this);
          browser.addActionListener(this);
     }
     public void actionPerformed(ActionEvent ae)
     {
          repaint();
     }
     // Display current selections.
     public void paint(Graphics g)
     {
          int idx[];
          msg = "Current OS: ";
          idx = os.getSelectedIndexes();
          for(int i=0; i<idx.length; i++)
          msg += os.getItem(idx[i]) + " ";
          g.drawString(msg, 6, 120);
          msg = "Current Browser: ";
          msg += browser.getSelectedItem();
          g.drawString(msg, 6, 140);
     }
}
```

Handling Scrollbars

```
// Demonstrate scroll bars.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SBDemo" width=300 height=200>
</applet>
*/
```

```
public class SBDemo extends Applet
implements AdjustmentListener, MouseMotionListener
{
     String msg = "";
     Scrollbar vertSB, horzSB;
     public void init()
     {
          int width = Integer.parseInt(getParameter("width"));
          int height = Integer.parseInt(getParameter("height"));
          vertSB = new Scrollbar(Scrollbar.VERTICAL,
               0, 1, 0, height);
          horzSB = new Scrollbar(Scrollbar.HORIZONTAL,
               0, 1, 0, width);
          add(vertSB);
          add(horzSB);
          // register to receive adjustment events
          vertSB.addAdjustmentListener(this);
          horzSB.addAdjustmentListener(this);
          addMouseMotionListener(this);
     }
     public void adjustmentValueChanged(AdjustmentEvent ae)
     {
          repaint();
     }
     // Update scroll bars to reflect mouse dragging.
     public void mouseDragged(MouseEvent me)
     {
          int x = me.getX();
          int y = me.getY();
          vertSB.setValue(y);
          horzSB.setValue(x);
          repaint();
     }
     // Necessary for MouseMotionListener
     public void mouseMoved(MouseEvent me)
     {
     }
     // Display current value of scroll bars.
     public void paint(Graphics g)
     {
          msg = "Vertical: " + vertSB.getValue();
          msg += ", Horizontal: " + horzSB.getValue();
          g.drawString(msg, 6, 160);
          // show current mouse drag position
          g.drawString("*", horzSB.getValue(),
          vertSB.getValue());
     }
}
```

Handling Text field

```
// Demonstrate text field.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/
public class TextFieldDemo extends Applet
implements ActionListener
{
     TextField name, pass;
     public void init()
     {
          Label namep = new Label("Name: ", Label.RIGHT);
          Label passp = new Label("Password: ", Label.RIGHT);
          name = new TextField(12);
          pass = new TextField(8);
          pass.setEchoChar('?');
          add(namep);
          add(name);
          add(passp);
          add(pass);
          // register to receive action events
          name.addActionListener(this);
          pass.addActionListener(this);
     }
     // User pressed Enter.
     public void actionPerformed(ActionEvent ae)
     {
          repaint();
     }
     public void paint(Graphics g)
     {
          g.drawString("Name: " + name.getText(), 6, 60);
          g.drawString("Selected text in name: "
               + name.getSelectedText(), 6, 80);
          g.drawString("Password: " + pass.getText(), 6, 100);
     }
}
```

Handling Menus

```
// Illustrate menus.
import java.awt.*;
```

```
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="MenuDemo1" width=250 height=250>
  </applet>
*/

// Create a subclass of Frame
class MenuFrame extends Frame {
  String msg = "";
  CheckboxMenuItem debug, test;

  MenuFrame(String title) {
    super(title);

    // create menu bar and add it to frame
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    // create the menu items
    Menu file = new Menu("File");
    MenuItem item1, item2, item3, item4, item5;
    file.add(item1 = new MenuItem("New..."));
    file.add(item2 = new MenuItem("Open..."));
    file.add(item3 = new MenuItem("Close"));
    file.add(item4 = new MenuItem("-"));
    file.add(item5 = new MenuItem("Quit..."));
    mbar.add(file);

    Menu edit = new Menu("Edit");
    MenuItem item6, item7, item8, item9;
    edit.add(item6 = new MenuItem("Cut"));
    edit.add(item7 = new MenuItem("Copy"));
    edit.add(item8 = new MenuItem("Paste"));
    edit.add(item9 = new MenuItem("-"));
    Menu sub = new Menu("Special");
    MenuItem item10, item11, item12;
    sub.add(item10 = new MenuItem("First"));
    sub.add(item11 = new MenuItem("Second"));
    sub.add(item12 = new MenuItem("Third"));
    edit.add(sub);

    // these are checkable menu items
    debug = new CheckboxMenuItem("Debug");
    edit.add(debug);
    test = new CheckboxMenuItem("Testing");
    edit.add(test);
```

```
    mbar.add(edit);

    // create an object to handle action and item events
    MyMenuHandler handler = new MyMenuHandler(this);
    // register it to receive those events
    item1.addActionListener(handler);
    item2.addActionListener(handler);
    item3.addActionListener(handler);
    item4.addActionListener(handler);
    item5.addActionListener(handler);
    item6.addActionListener(handler);
    item7.addActionListener(handler);
    item8.addActionListener(handler);
    item9.addActionListener(handler);
    item10.addActionListener(handler);
    item11.addActionListener(handler);
    item12.addActionListener(handler);
    debug.addItemListener(handler);
    test.addItemListener(handler);

    // create an object to handle window events
    MyWindowAdapter adapter = new MyWindowAdapter(this);
     // register it to receive those events
    addWindowListener(adapter);
  }

  public void paint(Graphics g) {
    g.drawString(msg, 10, 200);

    if(debug.getState())
      g.drawString("Debug is on.", 10, 220);
    else
      g.drawString("Debug is off.", 10, 220);

    if(test.getState())
      g.drawString("Testing is on.", 10, 240);
    else
      g.drawString("Testing is off.", 10, 240);
  }
}

class MyWindowAdapter extends WindowAdapter {
  MenuFrame menuFrame;
  public MyWindowAdapter(MenuFrame menuFrame) {
    this.menuFrame = menuFrame;
  }
  public void windowClosing(WindowEvent we) {
    menuFrame.setVisible(false);
```

```
  }
}

class MyMenuHandler implements ActionListener, ItemListener {
  MenuFrame menuFrame;
  public MyMenuHandler(MenuFrame menuFrame) {
    this.menuFrame = menuFrame;
  }
  // Handle action events
  public void actionPerformed(ActionEvent ae) {
      String msg = "You selected ";
      String arg = (String)ae.getActionCommand();
      if(arg.equals("New..."))
        msg += "New.";
      else if(arg.equals("Open..."))
        msg += "Open.";
      else if(arg.equals("Close"))
        msg += "Close.";
      else if(arg.equals("Quit..."))
        msg += "Quit.";
      else if(arg.equals("Edit"))
        msg += "Edit.";
      else if(arg.equals("Cut"))
        msg += "Cut.";
      else if(arg.equals("Copy"))
        msg += "Copy.";
      else if(arg.equals("Paste"))
        msg += "Paste.";
      else if(arg.equals("First"))
        msg += "First.";
      else if(arg.equals("Second"))
        msg += "Second.";
      else if(arg.equals("Third"))
        msg += "Third.";
      else if(arg.equals("Debug"))
        msg += "Debug.";
      else if(arg.equals("Testing"))
        msg += "Testing.";
      menuFrame.msg = msg;
      menuFrame.repaint();
  }
  // Handle item events
  public void itemStateChanged(ItemEvent ie) {
    menuFrame.repaint();
  }
}
// Create frame window.
public class MenuDemo1 extends Applet {
```

```
  Frame f;

  public void init() {
    f = new MenuFrame("Menu Demo");
    int width = Integer.parseInt(getParameter("width"));
    int height = Integer.parseInt(getParameter("height"));

    setSize(new Dimension(width, height));

    f.setSize(width, height);
    f.setVisible(true);
  }

  public void start() {
    f.setVisible(true);
  }

  public void stop() {
    f.setVisible(false);
  }
}
```

CardLayout

The CardLayout class is unique among the other layout managers in that it stores several different layouts. Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time. This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input. We can prepare the other layouts and have them hidden, ready to be activated when needed. CardLayout provides these two constructors:

```
CardLayout( )
CardLayout(int horz, int vert)
```

The first form creates a default card layout. The second form allows us to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type Panel. This panel must have CardLayout selected as its layout manager. The cards that form the deck are also typically objects of type Panel. Thus, we must create a panel that contains the deck and a panel for each card in the deck. Next, we add to the appropriate panel the components that form each card. We then add these panels to the panel for which CardLayout is the layout manager. Finally, we add this panel to the main applet panel. Once these steps are complete, we must provide some way for the user to select between cards. One common approach

is to include one push button for each card in the deck. When card panels are added to a panel, they are usually given a name. Thus, most of the time, we will use this form of add( ) when adding cards to a panel:

```
void add(Component panelObj, Object name);
```

Here, name is a string that specifies the name of the card whose panel is specified by *panelObj*. After we have created a deck, our program activates a card by calling one of the following methods defined by CardLayout:

```
void first(Container deck)
void last(Container deck)
void next(Container deck)
void previous(Container deck)
void show(Container deck, String cardName)
```

Here, *deck* is a reference to the container (usually a panel) that holds the cards, and *cardName* is the name of a card. Calling first( ) causes the first card in the deck to be shown. For showing the last card, call last( ) and for the next card, call next( ). To show the previous card, call previous( ). Both next( ) and previous( ) automatically cycle back to the top or bottom of the deck, respectively. The show( ) method displays the card whose name is passed in cardName. The following example creates a two-level card deck that allows the user to select an operating system. Windows-based operating systems are displayed in one card. Macintosh and Solaris are displayed in the other card.

The process of creating a card layout is visualized as below:

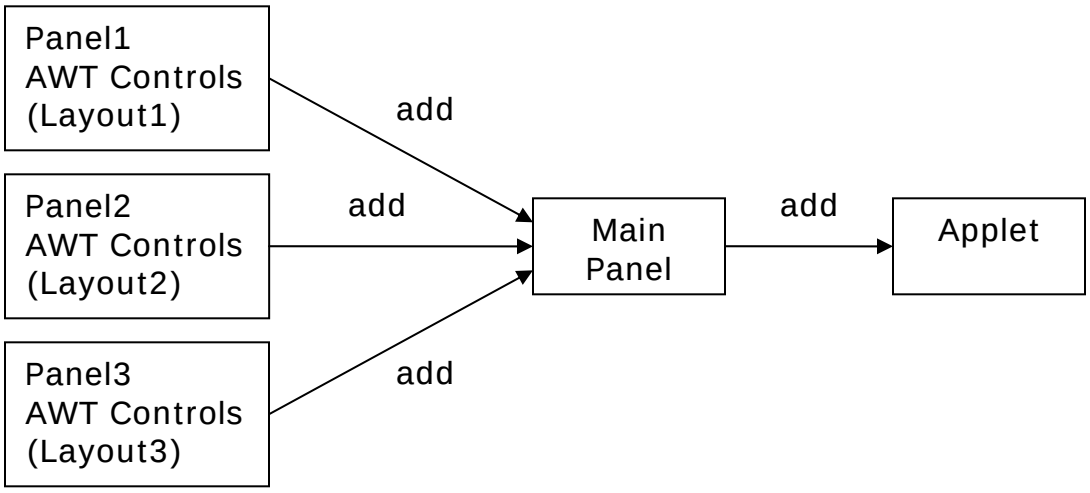


Fig. Creation of card layout

```
// Demonstrate CardLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
```

```
/*
  <applet code="CardLayoutDemo" width=300 height=100>
  </applet>
*/

public class CardLayoutDemo extends Applet
  implements ActionListener, MouseListener
  {

  Checkbox Win98, winNT, solaris, mac;
  Panel osCards;
  CardLayout cardLO;
  Button Win, Other;

  public void init()
  {
    Win = new Button("Windows");
    Other = new Button("Other");
    add(Win);
    add(Other);

    cardLO = new CardLayout();
    osCards = new Panel();
    osCards.setLayout(cardLO); // set panel layout to card layout

    Win98 = new Checkbox("Windows 98/XP", null, true);
    winNT = new Checkbox("Windows NT/2000");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("MacOS");

    // add Windows check boxes to a panel
    Panel winPan = new Panel();
    winPan.setLayout(new BorderLayout());
    winPan.add(Win98,BorderLayout.NORTH);
    winPan.add(winNT,BorderLayout.SOUTH);

    // Add other OS check boxes to a panel
    Panel otherPan = new Panel();
    otherPan.add(solaris);
    otherPan.add(mac);
    otherPan.setLayout(new GridLayout(2,2));

    // add panels to card deck panel
    osCards.add(winPan, "Windows");
    osCards.add(otherPan, "Other");

    // add cards to main applet panel
    add(osCards);
```

```
    // register to receive action events
    Win.addActionListener(this);
    Other.addActionListener(this);

    // register mouse events
    addMouseListener(this);
  }

  // Cycle through panels.
  public void mousePressed(MouseEvent me)
  {
    cardLO.next(osCards);
  }

  public void mouseClicked(MouseEvent me) {
  }
  public void mouseEntered(MouseEvent me) {
  }
  public void mouseExited(MouseEvent me) {
  }
  public void mouseReleased(MouseEvent me) {
  }

  public void actionPerformed(ActionEvent ae)
  {
    if(ae.getSource() == Win)
      cardLO.show(osCards, "Windows");
    else
      cardLO.show(osCards, "Other");
  }
}
```

Handling Events by Extending AWT Components

Java also allows us to handle events by subclassing AWT components. Doing so allows us to handle events in much the same way as they were handled under the original 1.0 version of Java. Of course, this technique is discouraged, because it has the same disadvantages of the Java 1.0 event model, the main one being inefficiency. In order to extend an AWT component, we must call the enableEvents( ) method of Component. Its general form is shown here:

```
protected final void enableEvents(long eventMask)
```

The eventMask argument is a bit mask that defines the events to be delivered to this component. The AWTEvent class defines int constants for making this mask. Several are shown here:

ACTION_EVENT_MASK	KEY_EVENT_MASK
ADJUSTMENT_EVENT_MASK	MOUSE_EVENT_MASK
COMPONENT_EVENT_MASK	MOUSE_MOTION_EVENT_MASK
CONTAINER_EVENT_MASK	MOUSE_WHEEL_EVENT_MASK
FOCUS_EVENT_MASK	TEXT_EVENT_MASK
INPUT_METHOD_EVENT_MASK	WINDOW_EVENT_MASK
ITEM_EVENT_MASK	

We must also override the appropriate method from one of our superclasses in order to process the event. Methods listed below most commonly used and the classes that provide them.

Event Processing Methods

Class	**Processing Methods**
Button	processActionEvent( )
Checkbox	processItemEvent( )
CheckboxMenuItem	processItemEvent( )
Choice	processItemEvent( )
Component	processComponentEvent( ), processFocusEvent( ), processKeyEvent( ), processMouseEvent( ), processMouseMotionEvent( ), processMouseWheelEvent ( )
List	processActionEvent( ), processItemEvent( )
MenuItem	processActionEvent( )
Scrollbar	processAdjustmentEvent( )
TextComponent	processTextEvent( )

Extending Button

The following program creates an applet that displays a button labeled “Test Button”. When the button is pressed, the string “action event: ” is displayed on the status line of the applet viewer or browser, followed by a count of the number of button presses. The program has one top-level class named ButtonDemo2 that extends Applet. A static integer variable named **i** is defined and initialized to zero. It records the number of button pushes. The init() method instantiates MyButton and adds it to the applet. MyButton is an inner class that extends Button. Its constructor uses super to pass the label of the button to the superclass constructor. It calls enableEvents( ) so that action events may be received by this object. When an action event is generated, processActionEvent( ) is called. That method displays a string on the status line

and calls processActionEvent( ) for the superclass. Because MyButton is an inner class, it has direct access to the showStatus( ) method of ButtonDemo2.

```
/*
* <applet code=ButtonDemo2 width=200 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ButtonDemo2 extends Applet
{
     MyButton myButton;
     static int i = 0;
     public void init()
     {
          myButton = new MyButton("Test Button");
          add(myButton);
     }
     class MyButton extends Button
     {
          public MyButton(String label)
          {
               super(label);
               enableEvents(AWTEvent.ACTION_EVENT_MASK);
          }
          protected void processActionEvent(ActionEvent ae)
          {
               showStatus("action event: " + i++);
               super.processActionEvent(ae);
          }
     }
}
```

Extending Checkbox

The following program creates an applet that displays three check boxes labeled “Item 1”, “Item 2”, and “Item 3”. When a check box is selected or deselected, a string containing the name and state of that check box is displayed on the status line of the applet viewer or browser.

The program has one top-level class named CheckboxDemo2 that extends Applet. Its init( ) method creates three instances of MyCheckbox and adds these to the applet. MyCheckbox is an inner class that extends Checkbox. Its constructor uses super to pass the label of the check box to the superclass constructor. It calls enableEvents( ) so that item events may be received by this object. When an item event is generated, processItemEvent( ) is called.

That method displays a string on the status line and calls processItemEvent( ) for the superclass.

```
/*
* <applet code=CheckboxDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class CheckboxDemo2 extends Applet
{
     MyCheckbox myCheckbox1, myCheckbox2, myCheckbox3;
     public void init()
     {
          myCheckbox1 = new MyCheckbox("Item 1");
          add(myCheckbox1);
          myCheckbox2 = new MyCheckbox("Item 2");
          add(myCheckbox2);
          myCheckbox3 = new MyCheckbox("Item 3");
          add(myCheckbox3);
     }
     class MyCheckbox extends Checkbox
     {
          public MyCheckbox(String label)
          {
               super(label);
               enableEvents(AWTEvent.ITEM_EVENT_MASK);
          }
          protected void processItemEvent(ItemEvent ie)
          {
               showStatus("Checkbox name/state: " + getLabel() +
               "/" + getState());
               super.processItemEvent(ie);
          }
     }
}
```

Extending a Check Box Group

The following program reworks the preceding check box example so that the check boxes form a check box group. Thus, only one of the check boxes may be selected at any time.

```
/*
* <applet code=CheckboxGroupDemo2 width=300 height=100>
* </applet>
*/
```

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class CheckboxGroupDemo2 extends Applet
{
     CheckboxGroup cbg;
     MyCheckbox myCheckbox1, myCheckbox2, myCheckbox3;
     public void init()
     {
          cbg = new CheckboxGroup();
          myCheckbox1 = new MyCheckbox("Item 1", cbg, true);
          add(myCheckbox1);
          myCheckbox2 = new MyCheckbox("Item 2", cbg, false);
          add(myCheckbox2);
          myCheckbox3 = new MyCheckbox("Item 3", cbg, false);
          add(myCheckbox3);
     }
     class MyCheckbox extends Checkbox
     {
          public   MyCheckbox(String   label,   CheckboxGroup   cbg,
          boolean flag)
          {
               super(label, cbg, flag);
               enableEvents(AWTEvent.ITEM_EVENT_MASK);
          }
          protected void processItemEvent(ItemEvent ie)
          {
               showStatus("Checkbox name/state: " + getLabel() +
               "/" + getState());
               super.processItemEvent(ie);
          }
     }
}
```

Extending Choice

The following program creates an applet that displays a choice list with items labeled “Red”, “Green”, and “Blue”. When an entry is selected, a string that contains the name of the color is displayed on the status line of the applet viewer or browser. There is one top-level class named ChoiceDemo2 that extends Applet. Its init( ) method creates a choice element and adds it to the applet. MyChoice is an inner class that extends Choice. It calls enableEvents( ) so that item events may be received by this object. When an item event is generated, processItemEvent( ) is called. That method displays a string on the status line and calls processItemEvent( ) for the superclass.

```
/*
* <applet code=ChoiceDemo2 width=300 height=100>
```

```
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ChoiceDemo2 extends Applet
{
     MyChoice choice;
     public void init()
     {
          choice = new MyChoice();
          choice.add("Red");
          choice.add("Green");
          choice.add("Blue");
          add(choice);
     }
     class MyChoice extends Choice
     {
          public MyChoice()
          {
               enableEvents(AWTEvent.ITEM_EVENT_MASK);
          }
          protected void processItemEvent(ItemEvent ie)
          {
               showStatus("Choice selection: " +
               getSelectedItem());
               super.processItemEvent(ie);
          }
     }
}
```

Extending List

The following program modifies the preceding example so that it uses a list instead of a choice menu. There is one top-level class named ListDemo2 that extends Applet. Its init( ) method creates a list element and adds it to the applet. MyList is an inner class that extends List. It calls enableEvents( ) so that both action and item events may be received by this object. When an entry is selected or deselected, processItemEvent( ) is called. When an entry is double-clicked, processActionEvent( ) is also called. Both methods display a string and then hand control to the superclass.

```
/*
* <applet code=ListDemo2 width=300 height=100>
* </applet>
*/
```

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ListDemo2 extends Applet
{
     MyList list;
     public void init()
     {
          list = new MyList();
          list.add("Red");
          list.add("Green");
          list.add("Blue");
          add(list);
     }
     class MyList extends List
     {
          public MyList()
          {
               enableEvents(AWTEvent.ITEM_EVENT_MASK |
               AWTEvent.ACTION_EVENT_MASK);
          }
          protected void processActionEvent(ActionEvent ae)
          {
               showStatus("Action event: " +
               ae.getActionCommand());
               super.processActionEvent(ae);
          }
          protected void processItemEvent(ItemEvent ie)
          {
               showStatus("Item event: " + getSelectedItem());
               super.processItemEvent(ie);
          }
     }
}
```

Extending Scrollbar

The following program creates an applet that displays a scroll bar. When this control is manipulated, a string is displayed on the status line of the applet viewer or browser. That string includes the value represented by the scroll bar. There is one top-level class named ScrollbarDemo2 that extends Applet. Its init( ) method creates a scroll bar element and adds it to the applet. MyScrollbar is an inner class that extends Scrollbar. It calls enableEvents( ) so that adjustment events may be received by this object. When the scroll bar is manipulated, processAdjustmentEvent( ) is called. When an entry is selected, processAdjustmentEvent( ) is called. It displays a string and then hands control to the superclass.

```
/*
* <applet code=ScrollbarDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ScrollbarDemo2 extends Applet
{
     MyScrollbar myScrollbar;
     public void init()
     {
          myScrollbar = new MyScrollbar(Scrollbar.HORIZONTAL,
          0, 1, 0, 100);
          add(myScrollbar);
     }
     class MyScrollbar extends Scrollbar
     {
          public MyScrollbar(int style, int initial, int thumb,
          int min, int max)
          {
               super(style, initial, thumb, min, max);
               enableEvents(AWTEvent.ADJUSTMENT_EVENT_MASK);
          }
          protected void processAdjustmentEvent(AdjustmentEvent ae)
          {
               showStatus("Adjustment event: " + ae.getValue());
               setValue(getValue());
               super.processAdjustmentEvent(ae);
          }
     }
}
```

References

1. **Java 2 the Complete Reference**,
   Fifth Edition by Herbert Schildt, 2001 Osborne McGraw Hill.
   Chapter 20: Event Handling
   Chapter 21: Introducing the AWT: Working with Windows, Graphics, and Text
   Chapter 22: Using AWT Controls, Layout Managers, and Menus
   (Most of the data is referred from this book)

2. **Learning Java,**
   3rd Edition , By Jonathan Knudsen, Patrick Niemeyer, O'Reilly, May 2005
   Chapter 19: Layout Managers

Chapter 02
Networking in Java

Lectures allotted: 08
Marks Given: 16

Contents:

2.1 Basics Socket overview, client/server, reserved sockets, proxy servers, internet addressing.
2.2 Java & the Net: The networking classes & interfaces
2.3 InetAddress class: Factory methods, instance method
2.4 TCP/IP Client Sockets,
2.5 TCP/IP Server Sockets
2.6 URL Format
2.7 URLConnection class
2.8 Data grams Data gram packets, Data gram server & Client

Introduction[Ref. 2]

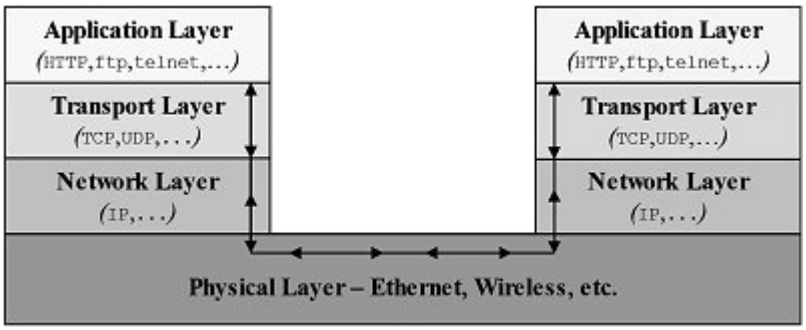
Sun Microsystems, the developer of Java having their motto as, 'The Network is Computer'. So they made the Java programming language, more appropriate for writing networked programs than, say, C++ or FORTRAN. What makes Java a good language for networking are the classes defined in the java.net package.

These networking classes encapsulate the socket paradigm pioneered in the Berkeley Software Distribution (BSD) from the University of California at Berkeley. No discussion of Internet networking libraries would be complete without a brief recounting of the history of UNIX and BSD sockets.

Networking Basics [Ref. 2]

Ken Thompson and Dennis Ritchie developed UNIX in concert with the C language at Bell Telephone Laboratories, in 1969. For many years, the development of UNIX remained in Bell Labs and in a few universities and research facilities that had the DEC-PDP machines it was designed to be run on. In 1978, Bill Joy was leading a project at Cal Berkeley to add many new features to UNIX, such as virtual memory and full-screen display capabilities. By early 1984, just as Bill was leaving to found Sun Microsystems, he shipped 4.2BSD, commonly known as Berkeley UNIX. 4.2 BSD came with a fast file system, reliable signals, inter-process communication, and, most important, networking. The networking support first found in 4.2 eventually became the de facto standard for the Internet. Berkeley's implementation of TCP/IP remains the primary standard for communications within the Internet. The socket paradigm for inter-process and network communication has also been widely adopted outside of Berkeley. Even Windows and the Macintosh started talking Berkeley sockets in the late 80s.

The OSI Reference Model [Ref. 5]



A formal OSI - Open System Interconnection - model has 7 layers but this one shows the essential layer definitions. Each layer has its own standardized protocols and applications programming interface (API), which refers to the functions, and their arguments and return values, called by the

next higher layer. Internally, the layers can be implemented in different ways as long as externally they obey the standard API.

For example, the Network Layer does not know if the Physical Layer is Ethernet or a wireless system because the device drivers respond to the function calls the same way. The Internet refers primarily to the Network Layer that implements the Internet Protocol (IP) and the Transport Layer that implements the Transmission Control Protocol (TCP). In fact, we often here people refer to the "TCP/IP" network rather than calling it the Internet.

The application layer also includes various protocols, such as FTP (File Transport Protocol) and HTTP (Hypertext Transfer Protocol) for the Web, that rely on the TCP/IP layers. Most users never look below the application layer. Most application programmers never work below the TCP/IP layers.

Socket Overview [Ref. 2]

A network socket is a lot like an electrical socket. Various plugs around the network have a standard way of delivering their payload. Anything that understands the standard protocol can plug into the socket and communicate. With electrical sockets, it doesn't matter if you plug in a lamp or a toaster; as long as they are expecting 50Hz, 115-volt electricity, the devices will work. Think how our electric bill is created. There is a meter some where between our house and the rest of the network. For each kilowatt of power that goes through that meter, we are billed. The bill comes to our address. So even though the electricity flows freely around the power grid, all of the sockets in our house have a particular address. The same idea applies to network sockets, except we talk about TCP/IP packets and IP addresses rather than electrons and street addresses. Internet Protocol (IP) is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination. Transmission Control Protocol (TCP) is a higher-level protocol that manages to robustly string together these packets, sorting and re-transmitting them as necessary to reliably transmit our data. A third protocol, User Datagram Protocol (UDP), sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

Client / Server [Ref. 2]

A server is anything that has some resource that can be shared. There are compute servers, which provide computing power; print servers, which manage a collection of printers; disk servers, which provide networked disk space; and web servers, which store web pages. A client is simply any other entity that wants to gain access to a particular server. The interaction between client and server is just like the interaction between a lamp and an electrical socket. The power grid of the house is the server, and the lamp is a power

client. The server is a permanently available resource, while the client is free to unplug after it is has been served.

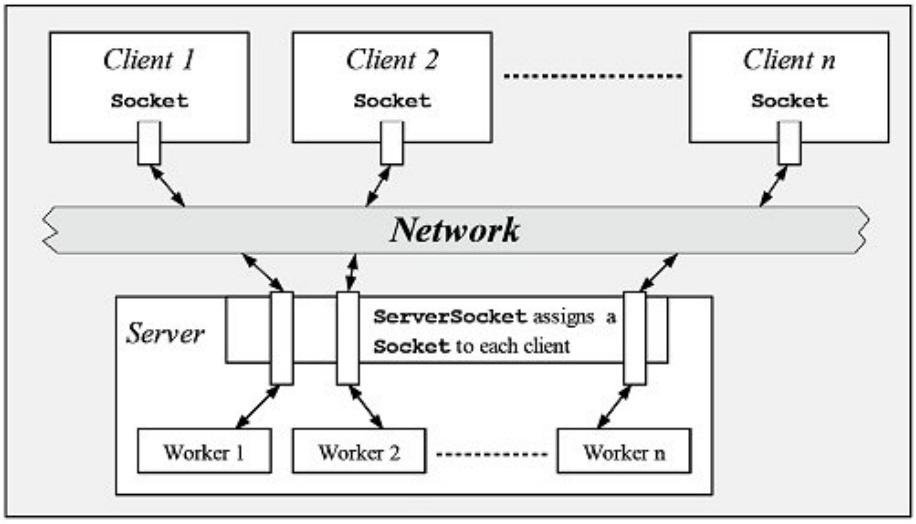


Fig. Client-Server Communication [Ref. 5]

In Berkeley sockets, the notion of a socket allows a single computer to serve many different clients at once, as well as serving many different types of information. This feat is managed by the introduction of a port, which is a numbered socket on a particular machine. A server process is said to listen to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique. To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O.

Reserved Sockets

Once connected, a higher-level protocol ensues, which is dependent on which port we are using. TCP/IP reserves the lower 1,024 ports for specific protocols. Many of these will seem familiar to us if we have spent any time surfing the Internet. Port number 21 is for FTP, 23 is for Telnet, 25 is for e-mail, 79 is for finger, 80 is for HTTP, 119 is for net-news and the list goes on. It is up to each protocol to determine how a client should interact with the port.

Table- Well-known port assignments [Ref. 1]

Protocol	Port	Protocol	Purpose
echo	7	TCP/UDP	Echo is a test protocol used to verify that two machines are able to connect by having one echo back the other's input.
discard	9	TCP/UDP	Discard is a less useful test protocol in which all data received by the server is ignored.

Table- Well-known port assignments [Ref. 1]			
Protocol	**Port**	**Protocol**	**Purpose**
daytime	13	TCP/UDP	Provides an ASCII representation of the current time on the server.
FTP data	20	TCP	FTP uses two well-known ports. This port is used to transfer files.
FTP	21	TCP	This port is used to send FTP commands like put and get.
SSH	22	TCP	Used for encrypted, remote logins.
telnet	23	TCP	Used for interactive, remote command-line sessions.
smtp	25	TCP	The Simple Mail Transfer Protocol is used to send email between machines.
time	37	TCP/UDP	A time server returns the number of seconds that have elapsed on the server since midnight, January 1, 1900, as a four-byte, signed, big-endian integer.
whois	43	TCP	A simple directory service for Internet network administrators.
finger	79	TCP	A service that returns information about a user or users on the local system.
HTTP	80	TCP	The underlying protocol of the World Wide Web.
POP3	110	TCP	Post Office Protocol Version 3 is a protocol for the transfer of accumulated email from the host to sporadically connected clients.
NNTP	119	TCP	Usenet news transfer; more formally known as the "Network News Transfer Protocol".
IMAP	143	TCP	Internet Message Access Protocol is a protocol for accessing mailboxes stored on a server.
RMI Registry	1099	TCP	The registry service for Java remote objects.

For example, HTTP is the protocol that web browsers and servers use to transfer hypertext pages and images. It is quite a simple protocol for a basic page-browsing web server. When a client requests a file from an HTTP server, an action known as a hit, it simply prints the name of the file in a special format to a predefined port and reads back the contents of the file. The server also responds with a status code number to tell the client whether the request can

be fulfilled and why. Here's an example of a client requesting a single file, /index.html, and the server replying that it has successfully found the file and is sending it to the client:

Server	**Client**
Listens to port 80	Connects to port 80
Accepts the connection	Writes GET /index.html HTTP/1.0\n\n.
Reads up until the second end-of-line (\n)	
Sees that GET is a known command and that HTTP/1.0 is a valid protocol version.	
Reads a local file called /index.html	
Writes HTTP/1.0 200 OK\n\n.	"200" means here comes the file
Copies the contents of the file into the socket.	Reads the contents of the file and displays it.
Hangs up.	Hangs up.

Proxy Servers [Ref. 2]

A proxy server speaks the client side of a protocol to another server. This is often required when clients have certain restrictions on which servers they can connect to. Thus, a client would connect to a proxy server, which did not have such restrictions, and the proxy server would in turn communicate for the client. A proxy server has the additional ability to filter certain requests or cache the results of those requests for future use. A caching proxy HTTP server can help reduce the bandwidth demands on a local network's connection to the Internet. When a popular web site is being hit by hundreds of users, a proxy server can get the contents of the web server's popular pages once, saving expensive internet work transfers while providing faster access to those pages to the clients.

Internet Addressing [Ref. 2]

Every computer on the Internet has an address. An Internet address is a number that uniquely identifies each computer on the Net. Originally, all Internet addresses consisted of 32-bit values. This address type was specified by IPv4 (Internet Protocol, version 4). However, a new addressing scheme, called IPv6 (Internet Protocol, version 6) has come into play. IPv6 uses a 128-bit value to represent an address. Although there are several reasons for and advantages to IPv6, the main one is that it supports a much larger address space than does IPv4. Fortunately, IPv6 is downwardly compatible with IPv4. Currently, IPv4 is by far the most widely used scheme, but this situation is likely to change over time.

Because of the emerging importance of IPv6, Java 2, version 1.4 has begun to add support for it. However, at the time of this writing, IPv6 is not supported by all environments. Furthermore, for the next few years, IPv4 will continue to be the dominant form of addressing. As mentioned, IPv4 is, loosely, a subset of IPv6, and the material contained in this chapter is largely applicable to both forms of addressing.

There are 32 bits in an IPv4 IP address, and we often refer to them as a sequence of four numbers between 0 and 255 separated by dots (.). This makes them easier to remember; because they are not randomly assigned they are hierarchically assigned. The first few bits define which class of network, lettered A, B, C, D, or E, the address represents. Most Internet users are on a class C network, since there are over two million networks in class C. The first byte of a class C network is between 192 and 224, with the last byte actually identifying an individual computer among the 256 allowed on a single class C network. This scheme allows for half a billion devices to live on class C networks.

Domain Naming Service (DNS) [Ref. 2]

The Internet wouldn't be a very friendly place to navigate if everyone had to refer to their addresses as numbers. For example, it is difficult to imagine seeing http://192.9.9.1/ at the bottom of an advertisement. Thankfully, a clearing house exists for a parallel hierarchy of names to go with all these numbers. It is called the Domain Naming Service (DNS). Just as the four numbers of an IP address describe a network hierarchy from left to right, the name of an Internet address, called its domain name, describes a machine's location in a name space, from right to left. For example, www.google.com is in the COM domain (reserved for commercial sites), it is called Google (after the company name), and www is the name of the specific computer that is Google's web server. www corresponds to the rightmost number in the equivalent IP address.

Java and the Net [Ref. 2]

Now that the stage has been set, let's take a look at how Java relates to all of these network concepts. Java supports TCP/IP both by extending the already established stream I/O interface and by adding the features required to build I/O objects across the network. Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, hence faster, point-to-point datagram-oriented model.

The Networking Classes and Interfaces [Ref. 2]

The classes contained in the java.net package are listed here:

Authenticator InetSocketAddress SocketImpl

ContentHandler	JarURLConnection	SocketPermission
DatagramPacket	MulticastSocket	URI
DatagramSocket	NetPermission	URL
DatagramSocketImpl	NetworkInterface	URLClassLoader
HttpURLConnection	PasswordAuthentication	URLConnection
InetAddress	ServerSocket	URLDecoder
Inet4Address	Socket	URLEncoder
Inet6Address	SocketAddress	URLStreamHandler

Some of these classes are to support the new IPv6 addressing scheme. Others provide some added flexibility to the original java.net package. Java 2, version 1.4 also added functionality, such as support for the new I/O classes, to several of the preexisting networking classes. The java.net package's interfaces are listed here:

ContentHandlerFactory	SocketImplFactory	URLStreamHandlerFactory
FileNameMap	SocketOptions	
DatagramSocketImplFactory		

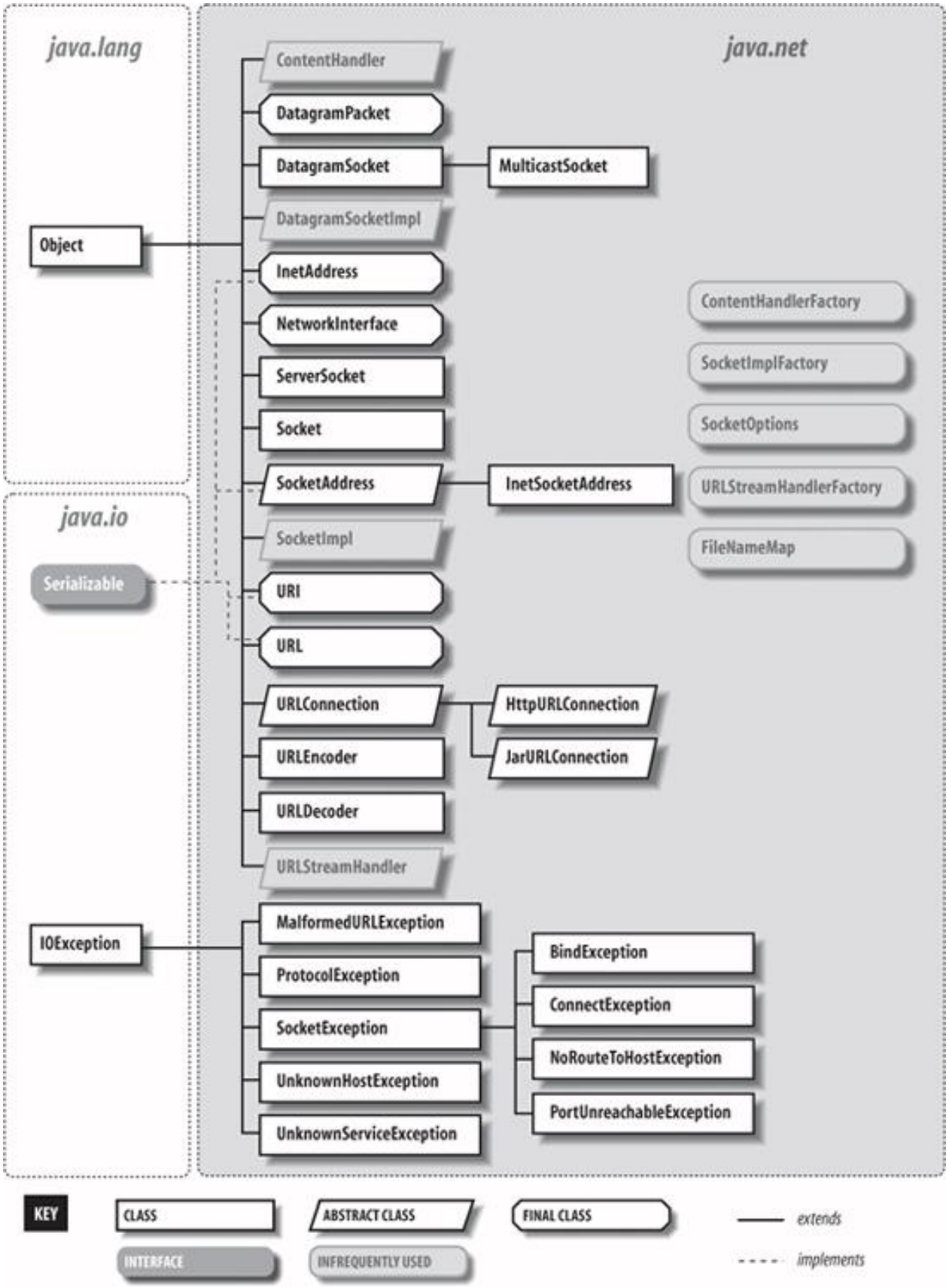


Fig. java.net package [Ref. 3]

InetAddress [Ref. 2]

Whether we are making a phone call, sending mail, or establishing a connection across the Internet, addresses are fundamental. The InetAddress class is used to encapsulate both the numerical IP address and the domain name for that address. We interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address. The InetAddress class hides the number inside. As of Java 2, version 1.4, InetAddress can handle both IPv4 and IPv6 addresses. This discussion assumes IPv4.

Factory Methods [Ref. 2]

The InetAddress class has no visible constructors. To create an InetAddress object, we have to use one of the available factory methods. Factory methods are merely a convention whereby static methods in a class return an instance of that class. This is done in lieu of overloading a constructor with various parameter lists when having unique method names makes the results much clearer. Three commonly used InetAddress factory methods are shown here.

```
static InetAddress getLocalHost( ) throws UnknownHostException
static InetAddress getByName(String hostName) throws
                          UnknownHostException
static InetAddress[ ] getAllByName(String hostName) throws
                          UnknownHostException
```

The getLocalHost( ) method simply returns the InetAddress object that represents the local host. The getByName( ) method returns an InetAddress for a host name passed to it. If these methods are unable to resolve the host name, they throw an UnknownHostException.

On the Internet, it is common for a single name to be used to represent several machines. In the world of web servers, this is one way to provide some degree of scaling. The getAllByName( ) factory method returns an array of InetAddresses that represent all of the addresses that a particular name resolves to. It will also throw an UnknownHostException if it can't resolve the name to at least one address. Java2, version1.4 also includes the factory method getByAddress(), which takes an IP address and returns an InetAddress object. Either an IPv4 or an IPv6 address can be used. The following example prints the addresses and names of the local machine and two well-known Internet web sites:

```
// Demonstrate InetAddress.
import java.net.*;
class InetAddressTest
{
```

```
     public static void main(String args[])
          throws UnknownHostException
     {
          InetAddress Address = InetAddress.getLocalHost();
          System.out.println(Address);
               Address = InetAddress.getByName("google.com");
          System.out.println(Address);
               InetAddress SW[] =
               InetAddress.getAllByName("www.yahoo.com");
          for (int i=0; i<SW.length; i++)
               System.out.println(SW[i]);
     }
}
```

```
itdept-server/192.168.1.75
google.com/209.85.171.100
www.yahoo.com/87.248.113.14
```

Instance Methods [Ref. 2]

The InetAddress class also has several other methods, which can be used on the objects returned by the methods just discussed. Here are some of the most commonly used.

`boolean equals(Object other)`
It returns true if this object has the same Internet address as other.

`byte[ ] getAddress( )`
It returns a byte array that represents the object's Internet address in network byte order.

`String getHostAddress( )`
It returns a string that represents the host address associated with the InetAddress object.

`String getHostName( )`
It returns a string that represents the host name associated with the InetAddress object.

`boolean isMulticastAddress( )`
It returns true if this Internet address is a multicast address. Otherwise, it returns false.

`String toString( )`
It returns a string that lists the host name and the IP address for convenience.

Internet addresses are looked up in a series of hierarchically cached servers. That means that our local computer might know a particular name-to-IP-address mapping automatically, such as for itself and nearby servers. For other names, it may ask a local DNS server for IP address information. If that server doesn't have a particular address, it can go to a remote site and ask for it. This can continue all the way up to the root server, called InterNIC (internic.net). This process might take a long time, so it is wise to structure our code so that we cache IP address information locally rather than look it up repeatedly.

TCP/IP Client Sockets [Ref. 2]

TCP/IP sockets are used to implement t reliable, bidirectional, persistent, point-to-point, and stream-based connections between hosts on the Internet. A socket can be used to connect Java's I/O system to other programs that may reside either on the local machine or on any other machine on the Internet.

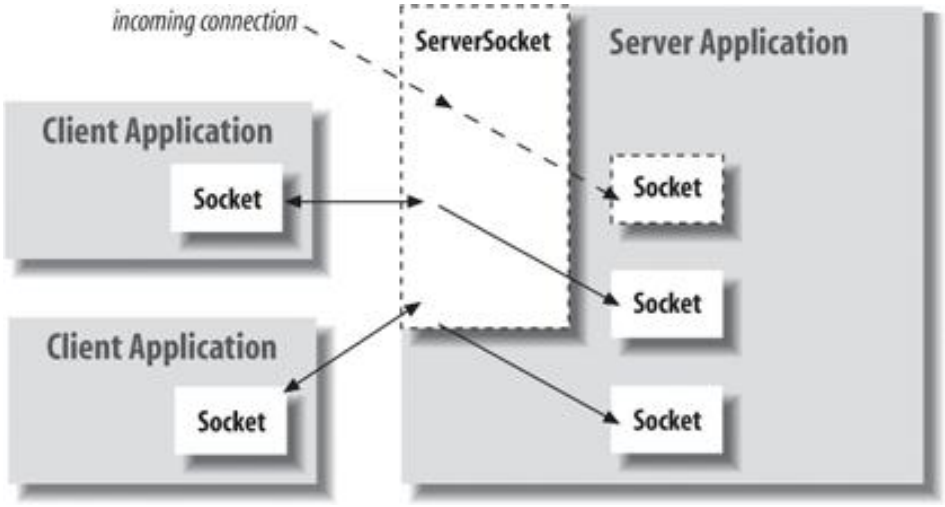


Fig. Clients and servers, Sockets and ServerSockets [Ref. 3]

Applets may only establish socket connections back to the host from which the applet was downloaded. This restriction exists because it would be dangerous for applets loaded through a firewall to have access to any arbitrary machine. There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The ServerSocket class is designed to be a listener, which waits for clients to connect before doing anything. The Socket class is designed to connect to server sockets and initiate protocol exchanges.

The creation of a Socket object implicitly establishes a connection between the client and server. There are no methods or constructors that explicitly expose the details of establishing that connection. Here are two constructors used to create client sockets:

`Socket(String hostName, int port) throws UnknownHostException, IOException`

Creates a socket connecting the local host to the named host and port; can throw an UnknownHostException or an IOException.

`Socket(InetAddress ipAddress, int port) throws UnknownHostException, IOException`

Creates a socket using a preexisting InetAddress object and a port; can throw an IOException. A socket can be examined at any time for the address and port information associated with it, by use of the following methods:

`InetAddress getInetAddress( )`

It returns the InetAddress associated with the Socket object.

`int getPort( )`

It returns the remote port to which this Socket object is connected.

`int getLocalPort( )`

It returns the local port to which this Socket object is connected.

Once the Socket object has been created, it can also be examined to gain access to the input and output streams associated with it. Each of these methods can throw an IOException if the sockets have been invalidated by a loss of connection on the Net. These streams are used exactly like the other I/O streams to send and receive data.

`InputStream getInputStream( )`

This returns the InputStream associated with the invoking socket.

`OutputStream getOutputStream( )`

This returns the OutputStream associated with the invoking socket.

Find out which of the first 1,024 ports seem to be hosting TCP servers on a specified host

```
import java.net.*;
import java.io.*;
public class LowPortScanner
{
     public static void main(String[] args)
     {
          String host = "localhost";
          for (int i = 1; i < 1024; i++)
          {
               try {
               Socket s = new Socket(host, i);
```

```
                System.out.println("There is a server on port " + i
                    + " of " + host);
      }
      catch (UnknownHostException ex)
      {
        System.err.println(ex);
        break;
      }
      catch (IOException ex)
      {
        // must not be a server on this port
      }
    } // end for
  }  // end main
}  // end PortScanner
```

Here's the output this program produces on local host. Results will vary, depending on which ports are occupied. As a rule, more ports will be occupied on a Unix workstation than on a PC or a Mac:

```
java LowPortScanner

There is a server on port 21 of localhost
There is a server on port 80 of localhost
There is a server on port 110 of localhost
There is a server on port 135 of localhost
There is a server on port 443 of localhost
```

A daytime protocol client [Ref. 1]

```
import java.net.*;
import java.io.*;
public class DaytimeClient
{
  public static void main(String[] args)
  {
    String hostname;
    try
    {
      Socket theSocket = new Socket(“localhost”, 13);
      InputStream timeStream = theSocket.getInputStream( );
      StringBuffer time = new StringBuffer( );
      int c;
      while ((c = timeStream.read( )) != -1) time.append((char) c);
      String timeString = time.toString( ).trim( );
      System.out.println("It is " + timeString + " at "
          + hostname);
```

```
    }  // end try
    catch (UnknownHostException ex)
    {
      System.err.println(ex);
    }
    catch (IOException ex)
    {
      System.err.println(ex);
    }
  }  // end main
} // end DaytimeClient
```

DaytimeClient reads the hostname of a daytime server from the command line and uses it to construct a new Socket that connects to port 13 on the server. Here the National Institute of Standards and Technology's time server at time.nist.gov is used as a host name. The client then calls theSocket.getInputStream( ) to get theSocket's input stream, which is stored in the variable timeStream. Since the daytime protocol specifies ASCII, DaytimeClient doesn't bother chaining a reader to the stream. Instead, it just reads the bytes into a StringBuffer one at a time, breaking when the server closes the connection as the protocol requires it to do. Here's what happens:

```
java DaytimeClient

It is 52956 03-11-13 04:45:28 00 0 0 706.3 UTC(NIST) * at
time.nist.gov
```

Whois [Ref. 2]

The very simple example that follows opens a connection to a whois port on the InterNIC server, sends the command-line argument down the socket, and then prints the data that is returned. InterNIC will try to lookup the argument as a registered Internet domain name, then send back the IP address and contact information for that site.

```
//Demonstrate Sockets.
import java.net.*;
import java.io.*;
class Whois
{
     public static void main(String args[]) throws Exception
     {
          int c;
          Socket s = new Socket("internic.net", 43);
          InputStream in = s.getInputStream();
          OutputStream out = s.getOutputStream();
          Stringstr=(args.length==0? "osborne.com":args[0])+"\n";
```

```
            byte buf[] = str.getBytes();
            out.write(buf);
            while ((c = in.read()) != -1)
                 System.out.print((char) c);
            s.close();
       }
}
```

If, for example, we obtained information about osborne.com, we'd get something similar to the following:

```
Whois Server Version 1.3
Domain names in the .com, .net, and .org domains can now be
registered with many different competing registrars. Go to
http://www.internic.net for detailed information.
Domain Name: OSBORNE.COM
Registrar: NETWORK SOLUTIONS, INC.
Whois Server: whois.networksolutions.com
Referral URL: http://www.networksolutions.com
Name Server: NS1.EPPG.COM
Name Server: NS2.EPPG.COM
Updated Date: 16-jan-2002
>> Last update of whois database: Thu, 25 Apr 2002 05:05:52 EDT <<
The Registry database contains ONLY .COM, .NET, .ORG, .EDU domains
and Registrars.
```

TCP/IP Server Sockets [Ref. 2]

Java has a different socket class that must be used for creating server applications. The ServerSocket class is used to create servers that listen for either local or remote client programs to connect to them on published ports. Since the Web is driving most of the activity on the Internet, this section develops an operational web (http) server.

ServerSockets are quite different from normal Sockets. When we create a ServerSocket, it will register itself with the system as having an interest in client connections. The constructors for ServerSocket reflect the port number that we wish to accept connections on and, optionally, how long we want the queue for said port to be. The queue length tells the system how many client connections it can leave pending before it should simply refuse connections. The default is 50.

The ServerSocket class contains everything needed to write servers in Java. It has constructors that create new ServerSocket objects, methods that listen for connections on a specified port, methods that configure the various server socket options, and the usual miscellaneous methods such as toString().

In Java, the basic life cycle of a server program is:

1. A new ServerSocket is created on a particular port using a ServerSocket() constructor.
2. The ServerSocket listens for incoming connection attempts on that port using its accept( ) method. accept( ) blocks until a client attempts to make a connection, at which point accept( ) returns a Socket object connecting the client and the server.
3. Depending on the type of server, either the Socket's getInputStream() method, getOutputStream( ) method, or both are called to get input and output streams that communicate with the client.
4. The server and the client interact according to an agreed-upon protocol until it is time to close the connection.
5. The server, the client, or both close the connection.
6. The server returns to step 2 and waits for the next connection.

The constructors might throw an IOException under adverse conditions. Here are the constructors:

`ServerSocket(int port) throws BindException, IOException`

It creates server socket on the specified port with a queue length of 50.

`ServerSocket(int port, int maxQueue) throws BindException, IOException`

This creates a server socket on the specified port with a maximum queue length of maxQueue.

`ServerSocket(int port, int maxQueue, InetAddress localAddress) throws IOException`

It creates a server socket on the specified port with a maximum queue length of maxQueue. On a multi-homed host, local Address specifies the IP address to which this socket binds. ServerSocket has a method called accept( ), which is a blocking call that will wait for a client to initiate communications, and then return with a normal Socket that is then used for communication with the client.

Scanner for the server ports: [Ref. 1]

```
import java.net.*;
import java.io.*;
public class LocalPortScanner
{
  public static void main(String[] args)
  {
    for (int port = 1; port <= 65535; port++)
    {
      try
      {
```

```
        // the next line will fail and drop into the catch block if
        // there is already a server running on the port
        ServerSocket server = new ServerSocket(port);
      }
      catch (IOException ex)
      {
        System.out.println("There is a server on port " + port
                           + ".");
      } // end catch
    } // end for
  }
}
```

Accepting and Closing Connections [Ref. 1]

A ServerSocket customarily operates in a loop that repeatedly accepts connections. Each pass through the loop invokes the accept( ) method. This returns a Socket object representing the connection between the remote client and the local server. Interaction with the client takes place through this Socket object. When the transaction is finished, the server should invoke the Socket object's close() method. If the client closes the connection while the server is still operating, the input and/or output streams that connect the server to the client throw an InterruptedIOException on the next read or write. In either case, the server should then get ready to process the next incoming connection. However, when the server needs to shut down and not process any further incoming connections, we should invoke the ServerSocket object's close( ) method.

```
public Socket accept( ) throws IOException
```

When server setup is done and we're ready to accept a connection, call the ServerSocket's accept() method. This method "blocks"; that is, it stops the flow of execution and waits until a client connects. When a client does connect, the accept( ) method returns a Socket object. We use the streams returned by this Socket's getInputStream( ) and getOutputStream( ) methods to communicate with the client. For example:

```
ServerSocket server = new ServerSocket(5776);
while (true)
{
  Socket connection = server.accept( );
  OutputStreamWriter out
   = new OutputStreamWriter(connection.getOutputStream( ));
  out.write("You've connected to this server. Bye-bye now.\r\n");
  connection.close( );
}
```

If we don't want the program to halt while it waits for a connection, put the call to accept( ) in a separate thread.

When exception handling is added, the code becomes somewhat more convoluted. It's important to distinguish between exceptions that should probably shut down the server and log an error message, and exceptions that should just close that active connection. Exceptions thrown by accept( ) or the input and output streams generally should not shut down the server. Most other exceptions probably should. To do this, we'll need to nest our try blocks.

Finally, most servers will want to make sure that all sockets they accept are closed when they're finished. Even if the protocol specifies that clients are responsible for closing connections, clients do not always strictly adhere to the protocol. The call to close( ) also has to be wrapped in a try block that catches an IOException. However, if we do catch an IOException when closing the socket, ignore it. It just means that the client closed the socket before the server could. Here's a slightly more realistic example:

```
try
{
  ServerSocket server = new ServerSocket(5776);
  while (true)
  {
    Socket connection = server.accept( );
    try
    {
      Writer out
      = new OutputStreamWriter(connection.getOutputStream( ));
      out.write("You've connected to this server. Bye-bye now.");
      out.flush( );
      connection.close( );
    }
    catch (IOException ex)
    {
     // This tends to be a transitory error for this one connection;
     // e.g. the client broke the connection early. Consequently,
     // we don't want to break the loop or print an error message.
     // However, we might choose to log this exception in an error log.
    }
    finally
    {
     // Guarantee that sockets are closed when complete.
     try
     {
       if (connection != null) connection.close( );
     }
     catch (IOException ex) {}
   }
}
```

```
catch (IOException ex)
{
  System.err.println(ex);
}
```

```
public void close( ) throws IOException
```

If we're finished with a server socket, we should close it, especially if the program is going to continue to run for some time. This frees up the port for other programs that may wish to use it. Closing a ServerSocket should not be confused with closing a Socket. Closing a ServerSocket frees a port on the local host, allowing another server to bind to the port; it also breaks all currently open sockets that the ServerSocket has accepted.

```
public InetAddress getInetAddress( )
```

This method returns the address being used by the server (the local host). If the local host has a single IP address (as most do), this is the address returned by InetAddress.getLocalHost( ). If the local host has more than one IP address, the specific address returned is one of the host's IP addresses. we can't predict which address we will get. For example:

```
ServerSocket httpd = new ServerSocket(80);
InetAddress ia = httpd.getInetAddress( );
```

If the ServerSocket has not yet bound to a network interface, this method returns null.

```
public int getLocalPort( )
```

The ServerSocket constructors allow us to listen on an unspecified port by passing 0 for the port number. This method lets us find out what port we're listening on.

Example: A Daytime server for daytime client: [Ref. 1]

```
public class DaytimeServer
{
  public static void main(String[] args)
  {
      try
   {
     ServerSocket server = new ServerSocket(13);
     Socket connection = null;
     while (true)
     {
       try
```

```
      {
        connection = server.accept( );
        Writer out = new
              OutputStreamWriter(connection.getOutputStream( ));
        Date now = new Date( );
        out.write(now.toString( ) +"\r\n");
        out.flush( );
        connection.close( );
      }
      catch (IOException ex) {}
      finally
      {
        try
        {
          if (connection != null) connection.close( );
        }
        catch (IOException ex) {}
      }
    }  // end while
  }  // end try
  catch (IOException ex)
  {
    System.err.println(ex);
  } // end catch
 } // end main
}
```

URL [Ref. 2]

The Web is a loose collection of higher-level protocols and file formats, all unified in a web browser. One of the most important aspects of the Web is that Tim Berners-Lee devised a scaleable way to locate all of the resources of the Net. Once we can reliably name anything and everything, it becomes a very powerful paradigm. The Uniform Resource Locator (URL) does exactly that.

The URL provides a reasonably intelligible form to uniquely identify or address information on the Internet. URLs are ubiquitous; every browser uses them to identify information on the Web. In fact, the Web is really just that same old Internet with all of its resources addressed as URLs plus HTML. Within Java's network class library, the URL class provides a simple, concise API to access information across the Internet using URLs.

Format [Ref. 2]

Two examples of URLs are http://www.rediff.com/ and http://www.rediff.com:80/index.htm/. A URL specification is based on four components. The first is the **protocol** to use, separated from the rest of the locator by a colon (:). Common protocols are http, ftp, gopher, and file,

although these days almost everything is being done via HTTP (in fact, most browsers will proceed correctly if we leave off the http:// from our URL specification). The second component is the **host name** or **IP address** of the host to use; this is delimited on the left by double slashes (//) and on the right by a slash (/) or optionally a colon (:). The third component, the **port number**, is an optional parameter, delimited on the left from the host name by a colon (:) and on the right by a slash (/). (It defaults to port 80, the predefined HTTP port; thus:80 is redundant.) The fourth part is the actual **file path**. Most HTTP servers will append a file named index.html or index.htm to URLs that refer directly to a directory resource. Thus, http://www.rediff.com/ is the same as http://www.rediff.com/index.htm. Java's URL class has several constructors, and each can throw a MalformedURLException. One commonly used form specifies the URL with a string that is identical to what you see displayed in a browser:

```
URL(String urlSpecifier)
```

The next two forms of the constructor allow you to break up the URL into its component parts:

```
URL(String protName, String hostName, int port, String path)
URL(String protName, String hostName, String path)
```

Another frequently used constructor allows us to use an existing URL as a reference context and then create a new URL from that context. Although this sounds a little contorted, it's really quite easy and useful.

```
URL(URL urlObj, String urlSpecifier)
```

In the following example, we create a URL to cric-info's news page and then examine its properties:

```
// Demonstrate URL. [Ref. 2]
import java.net.*;
class URLDemo
{
    public static void main(String args[])
        throws MalformedURLException
    {
        URL hp = new URL("http://content-
          ind.cricinfo.com/ci/content/current/story/news.html");
        System.out.println("Protocol: " + hp.getProtocol());
        System.out.println("Port: " + hp.getPort());
        System.out.println("Host: " + hp.getHost());
        System.out.println("File: " + hp.getFile());
        System.out.println("Ext:" + hp.toExternalForm());
```

```
    }
}
```

Output

```
Protocol: http
Port: -1
Host: content-ind.cricinfo.com
File: /ci/content/current/story/news.html
Ext:http://content-
ind.cricinfo.com/ci/content/current/story/news.html
```

Notice that the port is 1; this means that one was not explicitly set. Now that we have created a URL object, we want to retrieve the data associated with it. To access the actual bits or content information of a URL, we create a URLConnection object from it, using its openConnection( ) method, like this:

```
url.openConnection()
```

openConnection( ) has the following general form:

```
URLConnection openConnection( )
```

It returns a URLConnection object associated with the invoking URL object. It may throw an IOException.

URLConnection [Ref. 1]

URLConnection is an abstract class that represents an active connection to a resource specified by a URL. The URLConnection class has two different but related purposes. First, it provides more control over the interaction with a server (especially an HTTP server) than the URL class. With a URLConnection, we can inspect the header sent by the server and respond accordingly. We can set the header fields used in the client request. We can use a URLConnection to download binary files. Finally, a URLConnection lets us send data back to a web server with POST or PUT and use other HTTP request methods.

A program that uses the URLConnection class directly follows this basic sequence of steps:

1. Construct a URL object.
2. Invoke the URL object's openConnection( ) method to retrieve a URLConnection object for that URL.
3. Configure the URLConnection.
4. Read the header fields.
5. Get an input stream and read data.
6. Get an output stream and write data.
7. Close the connection.

We don't always perform all these steps. For instance, if the default setup for a particular kind of URL is acceptable, then we're likely to skip step 3. If we only want the data from the server and don't care about any meta-information, or if the protocol doesn't provide any meta-information, we'll skip step 4. If we only want to receive data from the server but not send data to the server, we'll skip step 6. Depending on the protocol, steps 5 and 6 may be reversed or interlaced.

The single constructor for the URLConnection class is protected:

```
protected URLConnection(URL url)
```

Consequently, unless we're sub-classing URLConnection to handle a new kind of URL (that is, writing a protocol handler), we can only get a reference to one of these objects through the openConnection( ) methods of the URL and URLStreamHandler classes. For example:

```
try {
  URL u = new URL("http://www.greenpeace.org/");
  URLConnection uc = u.openConnection( );
}
catch (MalformedURLException ex) {
  System.err.println(ex);
}
catch (IOException ex) {
  System.err.println(ex);
}
```

Reading Data from a Server [Ref. 1]

Here is the minimal set of steps needed to retrieve data from a URL using a URLConnection object:

1. Construct a URL object.
2. Invoke the URL object's openConnection( ) method to retrieve a URLConnection object for that URL.
3. Invoke the URLConnection's getInputStream( ) method.
4. Read from the input stream using the usual stream API.
5. The getInputStream() method returns a generic InputStream that lets you read and parse the data that the server sends.
6. public InputStream getInputStream( )

Example- Download a web page with a URLConnection [Ref. 1]

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

```
public class SourceViewer2 {
  public static void main (String[] args) {
    if  (args.length > 0) {
      try {
        //Open the URLConnection for reading
        URL u = new URL(args[0]);
        URLConnection uc = u.openConnection( );
        InputStream raw = uc.getInputStream( );
        InputStream buffer = new BufferedInputStream(raw);
        // chain the InputStream to a Reader
        Reader r = new InputStreamReader(buffer);
        int c;
        while ((c = r.read( )) != -1) {
          System.out.print((char) c);
        }
      }
      catch (MalformedURLException ex) {
        System.err.println(args[0] + " is not a parseable URL");
      }
      catch (IOException ex) {
        System.err.println(ex);
      }
    } //  end if
  } // end main
}  // end SourceViewer2
```

The differences between URL and URLConnection aren't apparent with just a simple input stream as in this example. The biggest differences between the two classes are: [Ref. 1]

1. URLConnection provides access to the HTTP header.
2. URLConnection can configure the request parameters sent to the server.
3. URLConnection can write data to the server as well as read data from the server.

Reading the Header [Ref. 1]

HTTP servers provide a substantial amount of information in the header that precedes each response. For example, here's a typical HTTP header returned by an Apache web server:

```
HTTP/1.1 200 OK
Date: Mon, 18 Oct 1999 20:06:48 GMT
Server: Apache/1.3.4 (Unix) PHP/3.0.6 mod_perl/1.17
Last-Modified: Mon, 18 Oct 1999 12:58:21 GMT
ETag: "1e05f2-89bb-380b196d"
Accept-Ranges: bytes
```

```
Content-Length: 35259
Connection: close
Content-Type: text/html
```

1. `public String getContentType( )`

This method returns the MIME content type of the data. It relies on the web server to send a valid content type.
For Example:
`text/plain, image/gif, application/xml, and image/jpeg.`

```
Content-type: text/html; charset=UTF-8
```
or
```
Content-Type: text/xml; charset=iso-2022-jp
```

2. `public int getContentLength( )`

The getContentLength( ) method tells us how many bytes there are in the content. Many servers send Content-length headers only when they're transferring a binary file, not when transferring a text file. If there is no Content-length header, getContentLength() returns -1. The method throws no exceptions. It is used when we need to know exactly how many bytes to read or when we need to create a buffer large enough to hold the data in advance.

3. `public long getDate( )`

The getDate( ) method returns a long that tells us when the document was sent, in milliseconds since midnight, Greenwich Mean Time (GMT), January 1, 1970. We can convert it to a java.util.Date. For example:

```
Date documentSent = new Date(uc.getDate( ));
```

This is the time the document was sent as seen from the server; it may not agree with the time on our local machine. If the HTTP header does not include a Date field, getDate( ) returns 0.

4. `public long getExpiration( )`

Some documents have server-based expiration dates that indicate when the document should be deleted from the cache and reloaded from the server. getExpiration( ) is very similar to getDate( ), differing only in how the return value is interpreted. It returns a long indicating the number of milliseconds after 12:00 A.M., GMT, January 1, 1970, at which point the document expires. If the HTTP header does not include an Expiration field, getExpiration( ) returns 0, which means 12:00 A.M., GMT, January 1, 1970. The only reasonable

interpretation of this date is that the document does not expire and can remain in the cache indefinitely.

5. public long getLastModified( )

The final date method, getLastModified( ), returns the date on which the document was last modified. Again, the date is given as the number of milliseconds since midnight, GMT, January 1, 1970. If the HTTP header does not include a Last-modified field (and many don't), this method returns 0.

Example:

```
import java.net.*;
import java.io.*;
import java.util.*;
public class HeaderViewer
{
  public static void main(String args[])
  {
      try
      {
        URL u = new URL("http://www.rediffmail.com/index.html");
        URLConnection uc = u.openConnection( );
        System.out.println("Content-type: " +
               uc.getContentType( ));
        System.out.println("Content-encoding: "
         + uc.getContentEncoding( ));
        System.out.println("Date: " + new Date(uc.getDate( )));
        System.out.println("Last modified: "
         + new Date(uc.getLastModified( )));
        System.out.println("Expiration date: "
         + new Date(uc.getExpiration( )));
        System.out.println("Content-length: " +
               uc.getContentLength( ));
      }  // end try
      catch (MalformedURLException ex)
      {
        System.out.println("I can't understand this URL...");
      }
      catch (IOException ex)
      {
        System.err.println(ex);
      }
      System.out.println( );
  }  // end main
}  // end HeaderViewer
```

Sample output:

```
Content-type: text/html
```

```
Content-encoding: null
Date: Mon Oct 18 13:54:52 PDT 1999
Last modified: Sat Oct 16 07:54:02 PDT 1999
Expiration date: Wed Dec 31 16:00:00 PST 1969
Content-length: -1
```

Sample output for: http://www.oreilly.com/graphics/space.gif

```
Content-type: image/gif
Content-encoding: null
Date: Mon Oct 18 14:00:07 PDT 1999
Last modified: Thu Jan 09 12:05:11 PST 1997
Expiration date: Wed Dec 31 16:00:00 PST 1969
Content-length: 57
```

Retrieving Arbitrary Header Fields [Ref. 1]

The last six methods requested specific fields from the header, but there's no theoretical limit to the number of header fields a message can contain. The next five methods inspect arbitrary fields in a header. Indeed, the methods of the last section are just thin wrappers over the methods discussed here; we can use these methods to get header fields that Java's designers did not plan for. If the requested header is found, it is returned. Otherwise, the method returns null.

```
public String getHeaderField(String name)
```

The getHeaderField() method returns the value of a named header field. The name of the header is not case-sensitive and does not include a closing colon. For example, to get the value of the Content-type and Content-encoding header fields of a URLConnection object uc, we could write:

```
String contentType = uc.getHeaderField("content-type");

String contentEncoding = uc.getHeaderField("content-encoding"));
```

To get the Date, Content-length, or Expires headers, you'd do the same:

```
String data = uc.getHeaderField("date");

String expires = uc.getHeaderField("expires");

String contentLength = uc.getHeaderField("Content-length");
```

These methods all return String, not int or long as the getContentLength( ), getExpirationDate(), getLastModified( ), and getDate( ) methods of the last

section did. If we're interested in a numeric value, convert the String to a long or an int.

Do not assume the value returned by getHeaderField() is valid. We must check to make sure it is non-null.

```
public String getHeaderFieldKey(int n)
```

This method returns the key (that is, the field name: for example, Content-length or Server) of the n^{th} header field. The request method is header zero and has a null key. The first header is one. For example, to get the sixth key of the header of the URLConnection uc, we would write:

```
String header6 = uc.getHeaderFieldKey(6);
```

```
public String getHeaderField(int n)
```

This method returns the value of the nth header field. In HTTP, the request method is header field zero and the first actual header is one. Example below uses this method in conjunction with getHeaderFieldKey( ) to print the entire HTTP header.

```
//Print the entire HTTP header
import java.net.*;
import java.io.*;
public class AllHeaders {
  public static void main(String args[]) {
    for (int i=0; i < args.length; i++) {
      try {
        URL u = new URL(args[i]);
        URLConnection uc = u.openConnection( );
        for (int j = 1; ; j++) {
          String header = uc.getHeaderField(j);
          if (header == null) break;
          System.out.println(uc.getHeaderFieldKey(j) + ": " +
               header);
        }  // end for
      }  // end try
      catch (MalformedURLException ex) {
        System.err.println(args[i] + " is not a URL I
               understand.");
      }
      catch (IOException ex) {
        System.err.println(ex);
      }
      System.out.println( );
    }  // end for
  }  // end main
```

```
}  // end AllHeaders
```

For example, here's the output when this program is run against http://www.oreilly.com:

```
java AllHeaders http://www.oreilly.com

Server: WN/1.15.1
Date: Mon, 18 Oct 1999 21:20:26 GMT
Last-modified: Sat, 16 Oct 1999 14:54:02 GMT
Content-type: text/html
Title: www.oreilly.com -- Welcome to O'Reilly &amp;amp; Associates!
-- computer  books, software, online publishing
Link: <mailto:webmaster@oreilly.com>; rev="Made"
```

Besides Date, Last-modified, and Content-type headers, this server also provides Server, Title, and Link headers. Other servers may have different sets of headers.

```
public long getHeaderFieldDate(String name, long default)
```

This method first retrieves the header field specified by the name argument and tries to convert the string to a long that specifies the milliseconds since midnight, January 1, 1970, GMT. getHeaderFieldDate() can be used to retrieve a header field that represents a date: for example, the Expires, Date, or Last-modified headers. To convert the string to an integer, getHeaderFieldDate() uses the parseDate( ) method of java.util.Date. The parseDate() method does a decent job of understanding and converting most common date formats, but it can be stumped—for instance, if we ask for a header field that contains something other than a date. If parseDate( ) doesn't understand the date or if getHeaderFieldDate( ) is unable to find the requested header field, getHeaderFieldDate( ) returns the default argument. For example:

```
Date expires = new Date(uc.getHeaderFieldDate("expires", 0));
long lastModified = uc.getHeaderFieldDate("last-modified", 0);
Date now = new Date(uc.getHeaderFieldDate("date", 0));
```

We can use the methods of the java.util.Date class to convert the long to a String.

```
public int getHeaderFieldInt(String name, int default)
```

This method retrieves the value of the header field name and tries to convert it to an int. If it fails, either because it can't find the requested header field or because that field does not contain a recognizable integer, getHeaderFieldInt( ) returns the default argument. This method is often used to

retrieve the Content-length field. For example, to get the content length from a URLConnection uc, you would write:

```
int contentLength = uc.getHeaderFieldInt("content-length", -1);
```

In this code fragment, getHeaderFieldInt( ) returns -1 if the Content-length header isn't present.

Configuring the Connection [Ref. 1]

The URLConnection class has seven protected instance fields that define exactly how the client makes the request to the server. These are:

```
protected URL     url;
protected boolean doInput = true;
protected boolean doOutput = false;
protected boolean allowUserInteraction =
                        defaultAllowUserInteraction;
protected boolean useCaches = defaultUseCaches;
protected long    ifModifiedSince = 0;
protected boolean connected = false;
```

For instance, if *doOutput* is true, we'll be able to write data to the server over this URLConnection as well as read data from it. If *useCaches* is false, the connection bypasses any local caching and downloads the file from the server afresh.

Since these fields are all protected, their values are accessed and modified via obviously named setter and getter methods:

```
public URL     getURL( )
public void    setDoInput(boolean doInput)
public boolean getDoInput( )
public void    setDoOutput(boolean doOutput)
public boolean getDoOutput( )
public void    setAllowUserInteraction(boolean allow)
public boolean getAllowUserInteraction( )
public void    setIfModifiedSince(long since)
public long    getIfModifiedSince( )
```

User Datagram Protocol (UDP) [Ref. 1]

The User Datagram Protocol (UDP) is an alternative protocol for sending data over IP that is very quick, but not reliable. That is, when you send UDP data, you have no way of knowing whether it arrived, much less whether different pieces of data arrived in the order in which you sent them. However, the pieces that do arrive generally arrive quickly.

The difference between TCP and UDP is often explained by analogy with the phone system and the post office. TCP is like the phone system. When we dial a number, the phone is answered and a connection is established between the two parties. As we talk, we know that the other party hears our words in the order in which we say them. If the phone is busy or no one answers, we find out right away. UDP, by contrast, is like the postal system. We send packets of mail to an address. Most of the letters arrive, but some may be lost on the way. The letters probably arrive in the order in which we sent them, but that's not guaranteed. The farther away we are from our recipient, the more likely it is that mail will be lost on the way or arrive out of order. If this is a problem, we can write sequential numbers on the envelopes, then ask the recipients to arrange them in the correct order and send us mail telling us which letters arrived so that we can resend any that didn't get there the first time. However, we and our correspondent need to agree on this protocol in advance. The post office will not do it for us.

Both the phone system and the post office have their uses. Although either one could be used for almost any communication, in some cases one is definitely superior to the other. The same is true of UDP and TCP.

Comparing TCP and UDP

No.	TCP	UDP
1	This Connection oriented protocol	This is connection-less protocol
2	The TCP connection is byte stream	The UDP connection is a message stream
3	It does not support multicasting and broadcasting	It supports broadcasting
4	It provides error control and flow control	The error control and flow control is not provided
5	TCP supports full duplex transmission	UDP does not support full duplex transmission
6	It is reliable service of data transmission	This is an unreliable service of data transmission
7	The TCP packet is called as segment	The UDP packet is called as user datagram.

Java's implementation of UDP is split into two classes: DatagramPacket and DatagramSocket. The DatagramPacket class stuffs bytes of data into UDP

packets called datagrams and lets us unstuff datagrams that we receive. A DatagramSocket sends as well as receives UDP datagrams. To send data, we put the data in a DatagramPacket and send the packet using a DatagramSocket. To receive data, we receive a DatagramPacket object from a DatagramSocket and then read the contents of the packet. The sockets themselves are very simple creatures. In UDP, everything about a datagram, including the address to which it is directed, is included in the packet itself; the socket only needs to know the local port on which to listen or send.

Datagrams [Ref. 1]

For most of our internetworking needs, we will be happy with TCP/IP-style networking. It provides a serialized, predictable, reliable stream of packet data. This is not without its cost, however. TCP includes many complicated algorithms for dealing with congestion control on crowded networks, as well as pessimistic expectations about packet loss. This leads to a somewhat inefficient way to transport data. Datagrams provide an alternative.

Datagrams are bundles of information passed between machines. They are somewhat like a hard throw from a well-trained but blindfolded catcher to the third baseman. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn't been damaged in transit or that whoever sent it is still there to receive a response.

Java implements datagrams on top of the UDP protocol by using two classes: The DatagramPacket object is the data container, while the DatagramSocket is the mechanism used to send or receive the DatagramPackets.

DatagramPacket [Ref. 1]

DatagramPacket uses different constructors depending on whether the packet will be used to send data or to receive data. This is a little unusual. Normally, constructors are overloaded to let us provide different kinds of information when we create an object, not to create objects of the same class that will be used in different contexts. In this case, all constructors take as arguments a byte array that holds the datagram's data and the number of bytes in that array to use for the datagram's data. When we want to receive a datagram, these are the only arguments we provide; in addition, the array should be empty. When the socket receives a datagram from the network, it stores the datagram's data in the DatagramPacket object's buffer array, up to the length we specified.

Constructors for receiving datagrams [Ref. 1]

These two constructors create new DatagramPacket objects for receiving data from the network:

```
public DatagramPacket(byte[] buffer, int length)
public DatagramPacket(byte[] buffer, int offset, int length)
```

When a socket receives a datagram, it stores the datagram's data part in buffer beginning at *buffer*[0] and continuing until the packet is completely stored or until length bytes have been written into the *buffer*. If the second constructor is used, storage begins at *buffer*[*offset*] instead. Otherwise, these two constructors are identical. *length* must be less than or equal to buffer.length-offset. If we try to construct a DatagramPacket with a length that will overflow the buffer, the constructor throws an *IllegalArgumentException*. This is a RuntimeException, so our code is not required to catch it. It is okay to construct a DatagramPacket with a length less than buffer.length-offset. In this case, at most the first length bytes of buffer will be filled when the datagram is received. For example, this code fragment creates a new DatagramPacket for receiving a datagram of up to 8,192 bytes:

```
byte[] buffer = new byte[8192];
DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
```

The constructor doesn't care how large the buffer is and would happily let us create a DatagramPacket with megabytes of data. However, the underlying native network software is less forgiving, and most native UDP implementations don't support more than 8,192 bytes of data per datagram.

Constructors for sending datagrams [Ref. 1]

These two constructors create new DatagramPacket objects for sending data across the network:

```
public DatagramPacket(byte[] data, int length,
                      InetAddress destination, int port)
public DatagramPacket(byte[] data, int offset, int length,
                      InetAddress destination, int port)
```

Each constructor creates a new DatagramPacket to be sent to another host. The packet is filled with *length* bytes of the data array starting at *offset* or 0 if offset is not used. If we try to construct a DatagramPacket with a *length* that is greater than *data.length*, the constructor throws an *IllegalArgumentException*. It's okay to construct a DatagramPacket object with an *offset* and a *length* that will leave extra, unused space at the end of the *data* array. In this case, only *length* bytes of *data* will be sent over the network. The

InetAddress object destination points to the host we want the packet delivered to; the int argument port is the port on that host.

For instance, this code fragment creates a new DatagramPacket filled with the data "This is a test" in ASCII. The packet is directed at port 7 (the echo port) of the host www.ibiblio.org:

```
String s = "This is a test";
byte[] data = s.getBytes("ASCII");
try {
  InetAddress ia = InetAddress.getByName("www.ibiblio.org");
  int port = 7;
  DatagramPacket dp = new
          DatagramPacket(data, data.length, ia, port);
  // send the packet...
}
catch (IOException ex){ }
```

The get Methods [Ref. 1]

DatagramPacket has six methods that retrieve different parts of a datagram: the actual data plus several fields from its header. These methods are mostly used for datagrams received from the network.

```
public InetAddress getAddress( )
```

The getAddress( ) method returns an InetAddress object containing the address of the remote host. If the datagram was received from the Internet, the address returned is the address of the machine that sent it (the source address). On the other hand, if the datagram was created locally to be sent to a remote machine, this method returns the address of the host to which the datagram is addressed (the destination address).

```
public int getPort( )
```

The getPort( ) method returns an integer specifying the remote port. If this datagram was received from the Internet, this is the port on the host that sent the packet. If the datagram was created locally to be sent to a remote host, this is the port to which the packet is addressed on the remote machine.

```
public SocketAddress getSocketAddress( )
```

The getSocketAddress() method returns a SocketAddress object containing the IP address and port of the remote host. As is the case for getInetAddress( ), if the datagram was received from the Internet, the address returned is the address of the machine that sent it (the source address).

```
public byte[] getData( )
```

The getData( ) method returns a byte array containing the data from the datagram. It's often necessary to convert the bytes into some other form of data before they'll be useful to our program. One way to do this is to change the byte array into a String using the following String constructor:

```
public String(byte[] buffer, String encoding)
```

If the datagram does not contain text, converting it to Java data is more difficult. One approach is to convert the byte array returned by getData() into a ByteArrayInputStream using this constructor:

```
public ByteArrayInputStream(byte[] buffer, int offset, int length)
```

'buffer' is the byte array to be used as an InputStream. It's important to specify the portion of the buffer that we want to use as an InputStream using the offset and length arguments.

```
public int getLength( )
```

The getLength( ) method returns the number of bytes of data in the datagram. This is not necessarily the same as the length of the array returned by getData( ), i.e., getData( ).length. The int returned by getLength( ) may be less than the length of the array returned by getData( ).

```
public int getOffset( )
```

This method simply returns the point in the array returned by getData( ) where the data from the datagram begins.

Following program uses all the methods covered above to print the information in the DatagramPacket. This example is a little artificial; because the program creates a DatagramPacket, it already knows what's in it. More often, we'll use these methods on a DatagramPacket received from the network, but that will have to wait for the introduction of the DatagramSocket class in the next section.

```
//Construct a DatagramPacket to receive data [Ref. 1]
import java.net.*;
public class DatagramExample1
{
  public static void main(String[] args)
  {
    String s = "This is a test.";
    byte[] data = s.getBytes( );
    try
```

```
    {
      InetAddress ia = InetAddress.getByName("www.ibiblio.org");
      int port = 7;
      DatagramPacket dp
       = new DatagramPacket(data, data.length, ia, port);
      System.out.println("This packet is addressed to "
       + dp.getAddress( ) + " on port " + dp.getPort( ));
      System.out.println("There are " + dp.getLength( )
       + " bytes of data in the packet");
      System.out.println(
      new String(dp.getData( ), dp.getOffset( ), dp.getLength( )));
    }
    catch (UnknownHostException e)
    {
      System.err.println(e);
    }
  }
}
```

Output:

```
This packet is addressed to www.ibiblio.org/152.2.254.81 on port 7
There are 15 bytes of data in the packet
This is a test.
```

The Set methods [Ref. 1]

`public void setData(byte[] data)`

The setData( ) method changes the payload of the UDP datagram. We might use this method if we are sending a large file (where large is defined as "bigger than can comfortably fit in one datagram") to a remote host. We could repeatedly send the same DatagramPacket object, just changing the data each time.

`public void setAddress(InetAddress remote)`

The setAddress( ) method changes the address a datagram packet is sent to. This might allow us to send the same datagram to many different recipients.

`public void setPort(int port)`

The setPort( ) method changes the port a datagram is addressed to.

`public void setLength(int length)`

The setLength( ) method changes the number of bytes of data in the internal buffer that are considered to be part of the datagram's data as opposed to merely unfilled space.

DatagramSocket [Ref. 1]

For sending or receiving a DatagramPacket, we must open a datagram socket. In Java, a datagram socket is created and accessed through the DatagramSocket class. The constructors are listed below:

```
public DatagramSocket( ) throws SocketException
public DatagramSocket(int port) throws SocketException
public DatagramSocket(int port, InetAddress interface)
     throws SocketException
```

The first constructor creates a socket that is bound to an anonymous port. Second constructor creates a socket that listens for incoming datagrams on a particular port, specified by the *port* argument. Third constructor is primarily used on multihomed hosts; it creates a socket that listens for incoming datagrams on a specific port and network interface. The port argument is the port on which this socket listens for datagrams. As with TCP sockets, we need to be root on a Unix system to create a DatagramSocket on a port below 1,024. The address argument is an InetAddress object matching one of the host's network addresses.

//The UDP Port Scanner [Ref. 1].

```
import java.net.*;
public class UDPPortScanner
{
  public static void main(String[] args)
  {
    for (int port = 0; port <= 2000; port++)
    {
      try
      {
        // the next line will fail and drop into the catch block if
        // there is already a server running on port i
        DatagramSocket server = new DatagramSocket(port);
        server.close( );
      }
      catch (SocketException ex)
      {
        System.out.println("There is a server on port " +
                    port + ".");
      } // end try
    } // end for
  }
}
```

Sample output:

```
There is a server on port 123.
```

```
There is a server on port 445.
There is a server on port 500.
There is a server on port 1900.
```

Sending and Receiving Datagrams [Ref. 1]

The primary task of the DatagramSocket class is to send and receive UDP datagrams. One socket can both send and receive. Indeed, it can send and receive to and from multiple hosts at the same time.

```
public void send(DatagramPacket dp) throws IOException
```

Once a DatagramPacket is created and a DatagramSocket is constructed, send the packet by passing it to the socket's send() method. For example, if theSocket is a DatagramSocket object and theOutput is a DatagramPacket object, send theOutput using theSocket like this:

```
theSocket.send(theOutput);
```

If there's a problem sending the data, an IOException may be thrown. However, this is less common with DatagramSocket than Socket or ServerSocket, since the unreliable nature of UDP means we won't get an exception just because the packet doesn't arrive at its destination. We may get an IOException if we are trying to send a larger datagram than the host's native networking software supports, but then again we may not. This method may also throw a SecurityException if the SecurityManager won't let you communicate with the host to which the packet is addressed.

```
public void receive(DatagramPacket dp) throws IOException
```

This method receives a single UDP datagram from the network and stores it in the preexisting DatagramPacket object *dp*. Like the accept( ) method in the ServerSocket class, this method blocks the calling thread until a datagram arrives. If our program does anything besides wait for datagrams, we should call receive() in a separate thread.

The datagram's buffer should be large enough to hold the data received. If not, receive( ) places as much data in the buffer as it can hold; the rest is lost. It may be useful to remember that the maximum size of the data portion of a UDP datagram is 65,507 bytes. (That's the 65,536-byte maximum size of an IP datagram minus the 20-byte size of the IP header and the 8-byte size of the UDP header.) Some application protocols that use UDP further restrict the maximum number of bytes in a packet; for instance, NFS uses a maximum packet size of 8,192 bytes.

If there's a problem receiving the data, an IOException may be thrown. In practice, this is rare. Unlike send( ), this method does not throw a

SecurityException if an applet receives a datagram from other than the applet host. However, it will silently discard all such packets.

Example: UDP Client [Ref. 1]

```
import java.net.*;
import java.io.*;
public class UDPDiscardClient
{
  public static void main(String[] args)
  {
    String hostname = "localhost";
    int port = 9;
    try
    {
      InetAddress server = InetAddress.getByName(hostname);
      BufferedReader userInput
       = new BufferedReader(new InputStreamReader(System.in));
      DatagramSocket theSocket = new DatagramSocket( );
      while (true)
      {
        String theLine = userInput.readLine( );
        if (theLine.equals(".")) break;
        byte[] data = theLine.getBytes( );
        DatagramPacket theOutput
         = new DatagramPacket(data, data.length, server, port);
        theSocket.send(theOutput);
      }  // end while
    }  // end try
    catch (UnknownHostException uhex) {
      System.err.println(uhex);
    }
    catch (SocketException se) {
      System.err.println(se);
    }
    catch (IOException ioex) {
      System.err.println(ioex);
    }
  }  // end main
}
```

//UDP Server

```
import java.net.*;
import java.io.*;
public class UDPDiscardServer
{
  public static void main(String[] args)
  {
    int port = 9;
```

```
    byte[] buffer = new byte[65507];
    try
    {
      DatagramSocket server = new DatagramSocket(port);
      DatagramPacket packet = new
                DatagramPacket(buffer, buffer.length);
      while (true)
      {
        try
        {
          server.receive(packet);
          String s = new String(packet.getData( ),
               0, packet.getLength( ));
          System.out.println(packet.getAddress( ) + " at port "
           + packet.getPort( ) + " says " + s);
          // reset the length for the next packet
          packet.setLength(buffer.length);
        }
        catch (IOException ex) {
          System.err.println(ex);
        }
       } // end while
    }  // end try
    catch (SocketException  ex)
    {
      System.err.println(ex);
    }  // end catch
  }  // end main
}
```

Sample outputs:

```
java  UDPDiscardClient
Hi
How
are
You?
Fine?

java  UDPDiscardServer
/127.0.0.1 at port 1027 says Hi
/127.0.0.1 at port 1027 says How
/127.0.0.1 at port 1027 says are
/127.0.0.1 at port 1027 says You?
/127.0.0.1 at port 1027 says Fine?
```

```
     public void close( )
```

Calling a DatagramSocket object's close( ) method frees the port occupied by that socket. For example:

```
try {
  DatagramSocket server = new DatagramSocket( );
  server.close( );
}
catch (SocketException ex) {
  System.err.println(ex);
}
```

It's never a bad idea to close a DatagramSocket when we're through with it; it's particularly important to close an unneeded socket if the program will continue to run for a significant amount of time.

```
public InetAddress getLocalAddress( )
```

A DatagramSocket's getLocalAddress( ) method returns an InetAddress object that represents the local address to which the socket is bound. It's rarely needed in practice. Normally, you either know or don't care which address a socket is listening to.

```
public void connect(InetAddress host, int port)
```

The connect( ) method doesn't really establish a connection in the TCP sense. However, it does specify that the DatagramSocket will send packets to and receive packets from only the specified remote *host* on the specified remote *port*. Attempts to send packets to a different *host* or *port* will throw an IllegalArgumentException. Packets received from a different *host* or a different *port* will be discarded without an exception or other notification.

```
public void disconnect( )
```

The disconnect( ) method breaks the "connection" of a connected DatagramSocket so that it can once again send packets to and receive packets from any host and port.

```
public int getPort( )
```

If and only if a DatagramSocket is connected, the getPort( ) method returns the remote port to which it is connected. Otherwise, it returns -1.

```
public InetAddress getInetAddress( )
```

If and only if a DatagramSocket is connected, the getInetAddress( ) method returns the address of the remote host to which it is connected. Otherwise, it returns null.

References

1. **Java Network Programming,**
   3rd Edition, By Elliotte Rusty Harold, O'Reilly, October 2004
   Chapter 2: Basic Networking Concepts
   Chapter 7: URLs and URIs
   Chapter 9: Sockets for Clients
   Chapter 10: Sockets for Servers
   Chapter 13: UDP Datagrams and Sockets
   Chapter 15: URL Connections
   (Most of the data is referred from this book)

2. **Java 2 the Complete Reference**,
   Fifth Edition by Herbert Schildt, 2001, Osborne McGraw Hill.
   Chapter 18: Networking

3. **Learning Java,**
   3rd Edition, By Jonathan Knudsen, Patrick Niemeyer, O'Reilly, May 2005
   Chapter 13: Network Programming

4. **A Laboratory Manual for Java Programming (1526)**,
   by Maharashtra State Board of Technical Education, Mumbai, 2004

5. http://www.particle.kth.se/~lindsey/JavaCourse/Book/index.html
   (A Good Site for Java Networking)

Chapter 03
Java Data-Base Connectivity

Lectures allotted: 08
Marks Given: 14

Contents:

3.1. Java as database front-end
Database client/server methodology
Two-Tier Database Design
Three-Tier Database Design
3.2. The JDBC API
The API Components
Limitations Using JDBC (Applications vs. Applets)
Security Considerations
A JDBC Database Example
JDBC Drivers
JDBC-ODBC Bridge
Current JDBC Drivers

Java as Database Front End [Ref.1, Ref.2]

Java offers several benefits to the developer creating a front-end application for a database server. Java is 'Write Once Run Everywhere' language. This means that Java programs may be deployed without recompilation on any computer architectures and operating systems that possesses a Java Virtual Machine.

In addition there is a cost associated with deployment and maintenance of the hardware and software of any system (client) the corporation owns Systems such as Windows PC, Macintosh and Unix desktop centric clients (fat clients) can cost corporations between $10,000 to $15,000 per installation seat. Java technology has made it possible for any company to use smaller system footprint. These systems are based on Java chip set and run any and all Java programs from built-in Java operating system.

Java based clients (thin clients) that operate with minimum of hardware resources, yet run the complete Java environment are expected to cost around $70 per seat. According to studies, saving for the corporations moving 10,000 fat clients to thin clients systems could be much as $100 million annually.

There are many industrial-strength DBMS available in the market. These include Oracle DB2, Sybase and many other popular brands. The challenge to Sun Microsystems faced in the late 1990s was to develop a way for Java developer to write a high level code that accesses all popular DBMSs.

The Sun Microsystems met the challenge in 1996 with the creation of JDBC driver for JDBC API. Both were created out of necessity, because until then Java wasn't industrial strength programming language since Java was unable to access the DBMS.

The JDBC driver developed by Sun wasn't driver at all. It was specification that described the detail functionality of JDBC driver. DBMS manufacturers and third-party vendors encouraged to build JDBC drivers that confirmed to Sun's specifications. Those firm that built JDBS drivers for their product could tap into growing Java applications market.

The specifications required a JDBC driver to be a translator that converted low-level proprietary DBMS messages to low-level messages understood by JDBC API and vice-versa. This meant that Java programmer could use high-level Java data-objects defined in the JDBC API to write a routine that interacted with the DBMS. Java data objects convert the routine into low-level message that conform to the JDBC driver specification and send them to the JDBC driver. The JDBC driver translates the routine into low-level messages that understood and processed by DBMS.

Database client-server methodology [Ref.2]

Relational databases are the most common DBMS. A main characteristic of a relational database is the absolute separation between physical and logical data. Data is accessed through the associated logical model to avoid supplying

physical storage locations and to reduce the limitations imposed by using physical information.

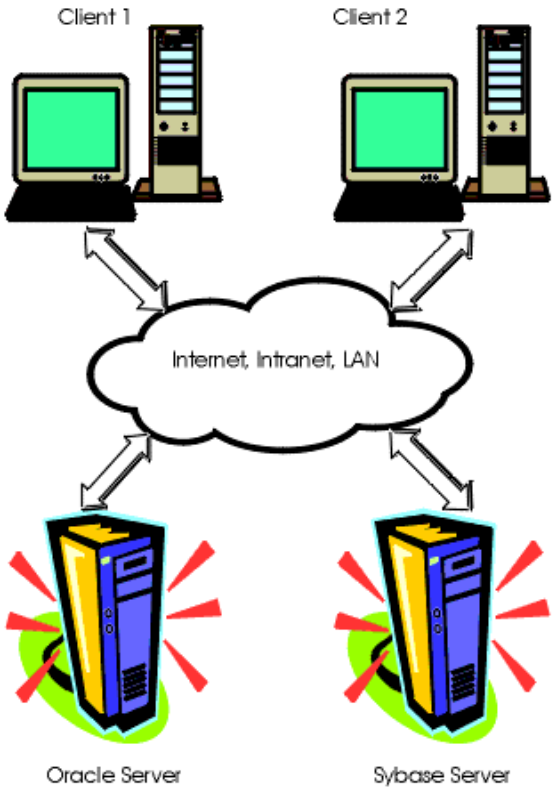


Fig. Database client/server architecture

Relational databases allow the definition of relations and integrity rules between data sets. E.F. Codd developed this model at the IBM San Jose Research Lab in the 1970s. A language to handle, define, and control data was also developed at the IBM lab: SQL. SQL stands for Structured Query Language. SQL is a query language that interacts with a DBMS. It allows data access without supplying physical access plans, data retrieval as sets of records, and the performing of complex computations on the data.

Software Architectures [Ref.2]

The first generation of client-server architectures is called two-tiered. It contains two active components: the client, which requests data, and the server, which delivers data. Basically, the application's processing is done separately for database queries and updates, and for user interface presentations. Usually the network binds the back end to the front end, although both tiers could be present on the same hardware.

For example, hundreds or thousands of airline seat reservation applications can connect to a central DBMS to request, insert, or modify data. While the clients process the graphics and data entry validation, the DBMS does

all the data processing. Actually, it is inadvisable to overload the database engine with data processing that is irrelevant to the server, thus some processing usually also happens on the clients. The typical client-server architecture is shown in Figure below:

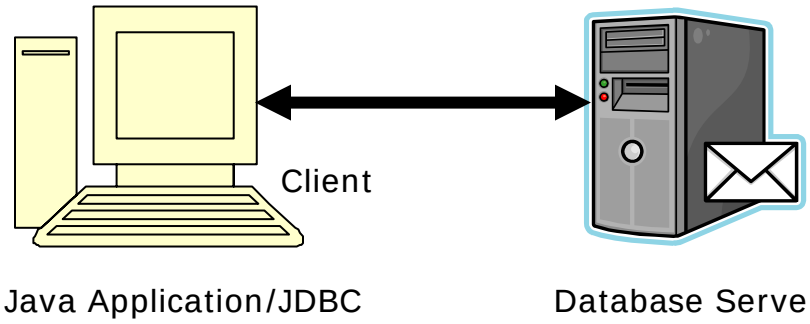


Fig. Two-tier client server architecture

The two tiers are often called as Application layer includes JDBC drivers, business logic and user interfaces whereas second layer i.e. Database layer consists of RDBMS server.

Advantages:

- It is simple in design.
- Client-side scripting offloads work onto the client

Drawbacks:

- Fat client.
- It is inflexible.

Although the two-tiered architecture is common, another design is starting to appear more frequently. To avoid embedding the application's logic at both the database side and the client side, a third software tier may be inserted. In three-tiered architectures, most of the business logic is frozen in the middle tier. In this architecture, when the business activity or business rules change, only the middleware must be modified. Figure below illustrates the three-tier architecture.

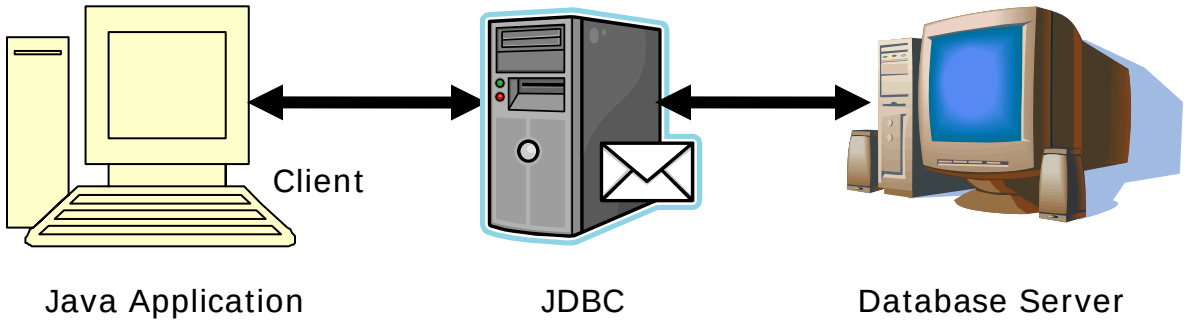


Fig. Three-tier client/server architecture

Advantages:

- Flexible: It can change one part without affecting others.

- It can connect to different databases without changing code.
- Specialization: presentation / business logic / data management.
- It can cache queries.
- It can implement proxies and firewalls.

Drawbacks:

- Higher complexity
- Higher maintenance
- Lower network efficiency
- More parts to configure (and buy)

What is JDBC? [Ref.5]

The JDBC stands for Java Database Connectivity. What is this JDBC besides a nifty acronym? It refers to several things, depending on context:

- It's a specification for using data sources in Java applets and applications.
- It's an API for using low-level JDBC drivers.
- It's an API for creating the low-level JDBC drivers, which do the actual connecting/transacting with data sources.
- It's based on the X/Open SQL Call Level Interface (CLI) that defines how client/server interactions are implemented for database systems.

The JDBC defines every aspect of making data-aware Java applications and applets. The low-level JDBC drivers perform the database-specific translation to the high-level JDBC interface. This interface is used by the developer so he doesn't need to worry about the database-specific syntax when connecting to and querying different databases. The JDBC is a package, much like other Java packages such as java.awt. It's not currently a part of the standard Java Developer's Kit (JDK) distribution, but it is slated to be included as a standard part of the general Java API as the java.sql package. Soon after its official incorporation into the JDK and Java API, it will also become a standard package in Java-enabled Web browsers, though there is no definite timeframe for this inclusion. The exciting aspect of the JDBC is that the drivers necessary for connection to their respective databases do not require any pre-installation on the clients: A JDBC driver can be downloaded along with an applet!

The JDBC project was started in January of 1996, and the specification was frozen in June of 1996. Javasoft sought the input of industry database vendors so that the JDBC would be as widely accepted as possible when it was ready for release. And, as we can see from this list of vendors who have already endorsed the JDBC, it's sure to be widely accepted by the software industry:

- Borland International, Inc.
- Bulletproof
- Cyber SQL Corporation
- DataRamp
- Dharma Systems, Inc.

- Gupta Corporation
- IBM's Database 2 (DB2)
- Imaginary (mSQL)
- Informix Software, Inc.
- Intersoft
- Intersolv
- Object Design, Inc.
- Open Horizon
- OpenLink Software
- Oracle Corporation
- Persistence Software
- Presence Information Design
- PRO-C, Inc.
- Recital Corporation
- RogueWave Software, Inc.
- SAS Institute, Inc. ™
- SCO
- Sybase, Inc.
- Symantec
- Thunderstone
- Visigenic Software, Inc.
- WebLogic, Inc.
- XDB Systems, Inc.

The JDBC is heavily based on the ANSI SQL-92 standard, which specifies that a JDBC driver should be SQL-92 entry-level compliant to be considered a 100 percent JDBC-compliant driver. This is not to say that a JDBC driver has to be written for an SQL-92 database; a JDBC driver can be written for a legacy database system and still function perfectly. Even though the driver does not implement every single SQL-92 function, it is still a JDBC driver. This flexibility will be a major selling point for developers who are bound to legacy database systems but who still want to extend their client applications.

What is ODBC? [Ref.5]

Many database servers use vendor-specific protocols. This means that database client has to learn a new language to talk to different database server. However, Microsoft established a common standard for communicating with databases called Open Database Connectivity that is, ODBC. Until ODBC, most database clients were server-specific. ODBC drivers abstract away vendor-specific protocols, providing a common API to database clients. By writing our database clients to the ODBC API, we enable our programs to access more database servers.

JDBC provides a common database programming API for Java programs. However, JDBC drivers do not directly communicate with as many database

products as ODBC drivers. Instead, many JDBC drivers communicate with database using ODBC. In fact, one of first JDBC drivers was the JDBC-ODBC bridge driver developed by JavaSoft.

ODBC is a C language API. C uses pointers and other dangerous programming constructs that Java does not support. A Java version of ODBC would require a significant rewrite of ODBC API. ODBC drivers must be installed on client machines. This means that the applet access to databases would be constrained by the requirements to download and install JDBC driver. A pure Java solution allows JDBC drivers to be automatically down-loaded and installed along with the applet. This greatly simplifies database access for applet users.

The JDBC Structure [Ref. 2]

The JDBC is two-dimensional. The reasoning for the split is to separate the low-level programming from the high-level application interface. The low-level programming is the JDBC driver. The idea is that database vendors and third-party software vendors will supply pre-built drivers for connecting to different databases. JDBC drivers are quite flexible: They can be local data sources or remote database servers. The implementation of the actual connection to the data source/database is left entirely to the JDBC driver. The structure of the JDBC includes these key concepts:

- The goal of the JDBC is a DBMS independent interface, a “generic SQL database access framework,” and a uniform interface to different data sources.
- The programmer writes only one database interface; using JDBC, the program can access any data source without recoding.

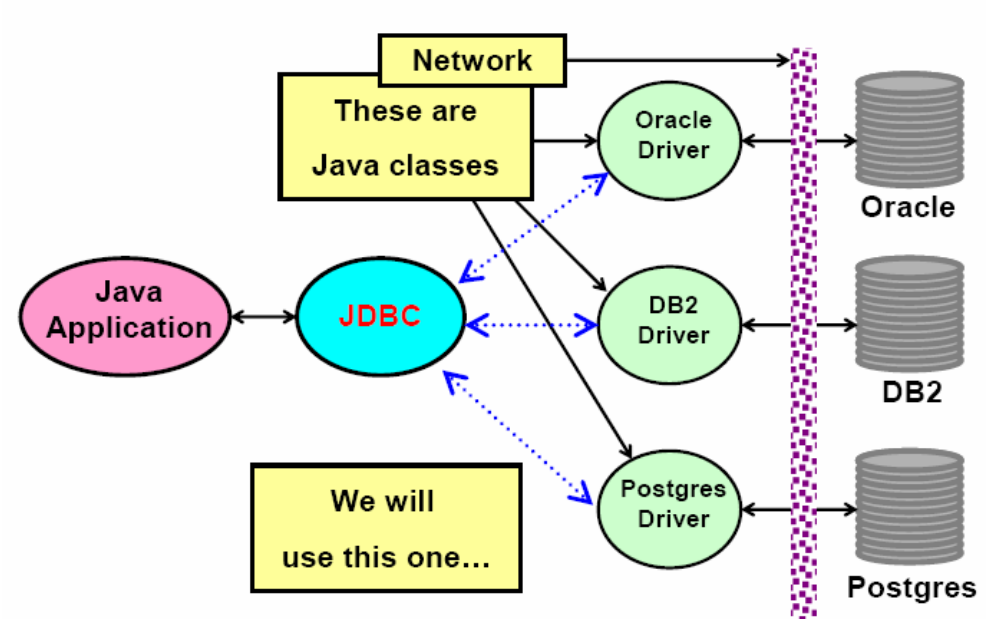


Fig. The JDBC Architecture

JDBC drivers [Ref.1]

Sun has defined four categories of JDBC drivers. The categories delineate the differences in architecture for the drivers. One difference between architectures lies in whether a given driver is implemented in native code or in Java code. Native code means whatever machine code is supported by a particular hardware configuration. For example, a driver may be written in C and then compiled to run on a specific hardware platform. Another difference lies in how the driver makes the actual connection to the database. The four driver types are as follows:

Type 1 Driver: JDBC/ODBC Bridge

This type uses bridge technology to connect a Java client to a third-party API such as Open DataBase Connectivity (ODBC). Sun's JDBC-ODBC bridge is an example of a Type 1 driver. These drivers are implemented using native code. This driver connects Java to a Microsoft ODBC (Open Database Connectivity) data source.

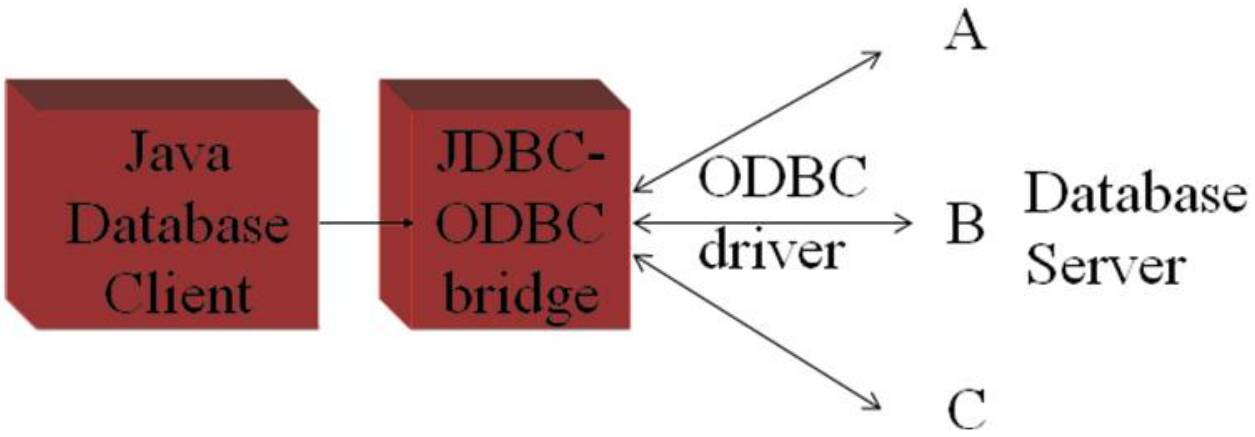


Fig. Type 1 Driver

The Java 2 Software Development Kit from Sun Microsystems, Inc. includes the JDBC-to-ODBC bridge driver (sun.jdbc.odbc.JdbcOdbcDriver). This driver typically requires the ODBC driver to be installed on the client computer and normally requires configuration of the ODBC data source. The bridge driver was introduced primarily to allow Java programmers to build data-driven Java applications before the database vendors had Type 3 and Type 4 drivers.

Type 2 Driver: Native API Driver

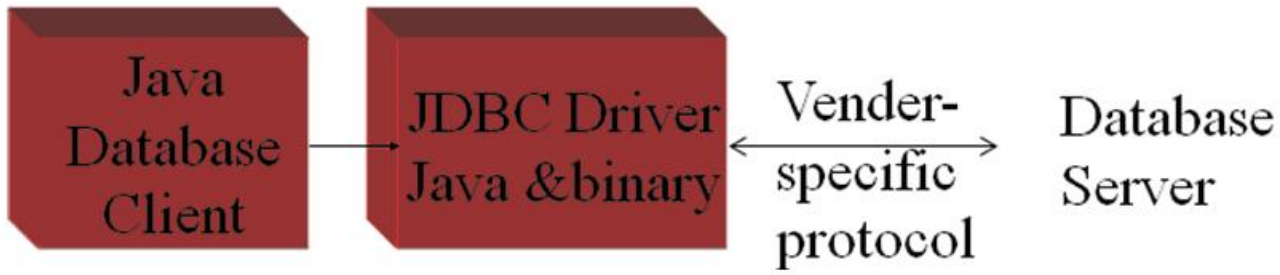


Fig. Type 2 Driver

This type of driver wraps a native API with Java classes. The Oracle Call Interface (OCI) driver is an example of a Type 2 driver. Because a Type 2 driver is implemented using local native code, it is expected to have better performance than a pure Java driver. These drivers enable JDBC programs to use database-specific APIs (normally written in C or C++) that allow client programs to access databases via the Java Native Interface. This driver type translates JDBC into database-specific code. Type 2 drivers were introduced for reasons similar to the Type 1 ODBC bridge driver.

Type 3 Driver: Network Protocol, Pure Java Driver

These drivers take JDBC requests and translate them into a network protocol that is not database specific. These requests are sent to a server, which translates the database requests into a database-specific protocol.

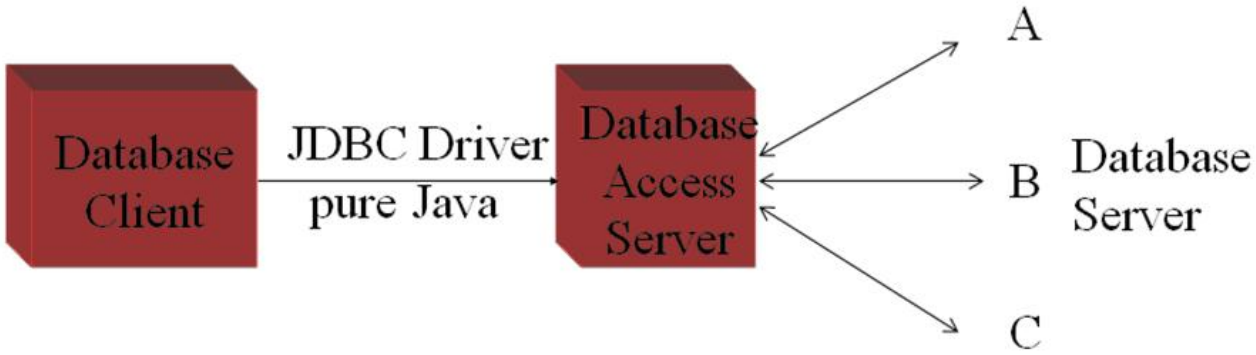


Fig. Type 3 Driver

This type of driver communicates using a network protocol to a middle-tier server. The middle tier in turn communicates to the database. Oracle does not provide a Type 3 driver. They do, however, have a program called Connection Manager that, when used in combination with Oracle's Type 4 driver, acts as a Type 3 driver in many respects.

Type 4 Driver: Native Protocol, Pure Java Driver

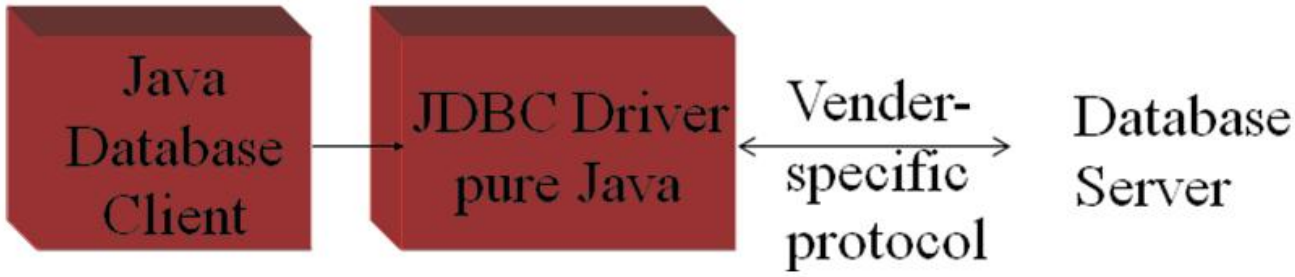


Fig. Type 4 Driver

These convert JDBC requests to database-specific network protocols, so that Java programs can connect directly to a database. This type of driver, written entirely in Java, communicates directly with the database. No local native code is required. Oracle's thin driver is an example of a Type 4 driver.

The JDBC API [Ref. 2]

The JDBC API is contained in two packages named java.sql and javax.sql. The java.sql package contains core Java objects of JDBC API. There are two distinct layers within the JDBC API: the application layer, which database-application developers use and driver layer which the drivers vendors implement. The connection between application and driver layers is illustrated in figure below:

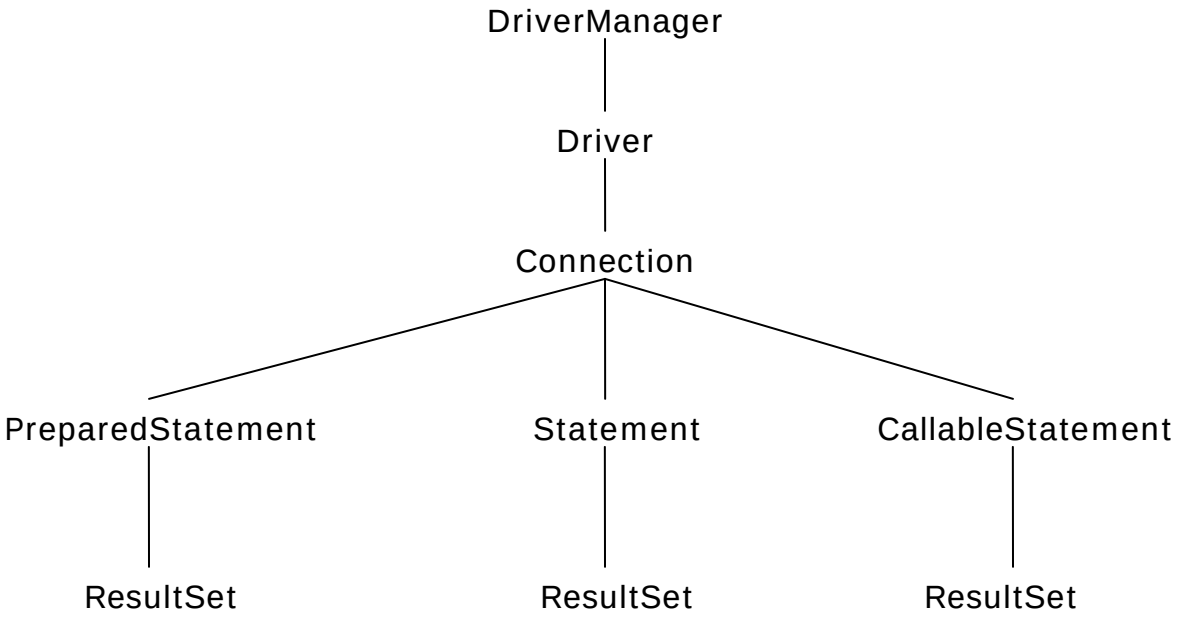


Fig. The JDBC API

There are four main interfaces that every driver layer must implement and one class that bridges the Application and driver layers. The four interfaces are Driver, Connection, Statement and ResultSet. The Driver interface

implementation is where the connection to the database is made. In most applications, Driver is accessed through DriverManager class.

The JDBC process [Ref. 1]

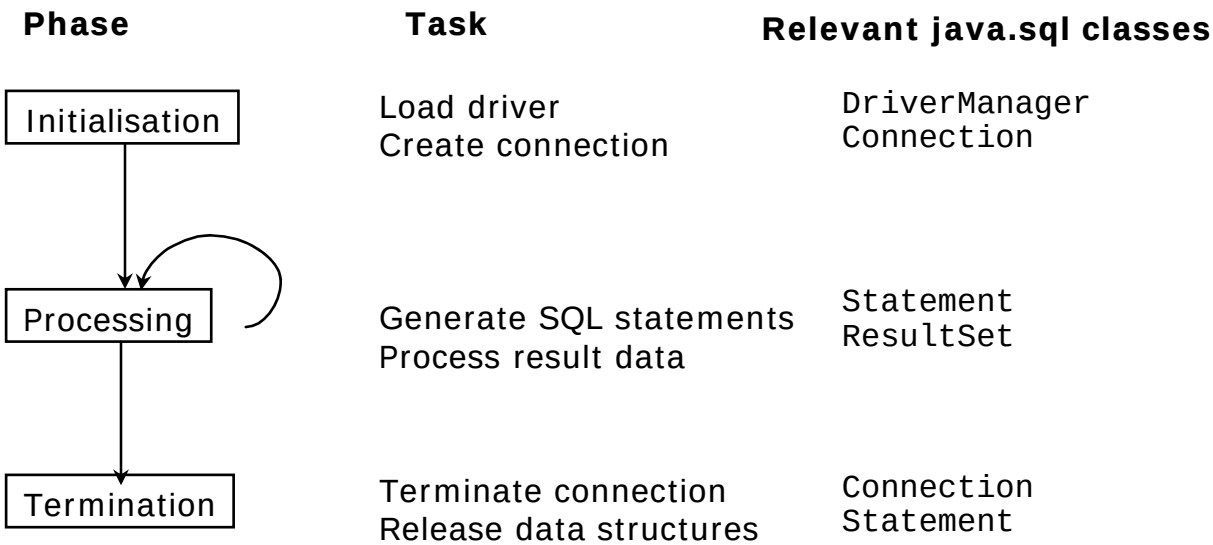
Accessing JDBC / ODBC Bridge with the database

Before actual performing the Java database application, we associate the connection of database source using JDBC – ODBC Bridge. The steps are as follows:

1. Go to Control Panel -> Administrative Tools -> Data Sources.
2. Open Data Sources ODBC icon.

3. Select the tab with heading “User DSN”.
4. Click on ‘Add’ button.
5. Select the appropriate driver as per the database to be used. (e.g. Microsoft ODBC driver for Oracle to access Oracle Database
6. Click finish button and the corresponding ODBC database setup window will appear.
7. Type DSN name and provide the required information such as user name and password for the database (.mdb files) of Microsoft Access Database etc. and click on OK button.
8. Our DSN name will get appeared in user data sources.

There are six different steps to use JDBC in our Java application program. These can be shown diagrammatically as below:



1. Load the driver

2. Define and establish the Connection
3. Create a Statement object
4. Execute a query
5. Process the results
6. Close the connection

Loading the JDBC driver

The JDBC drivers must be loaded before the Java application connects to the DBMS. The `Class.forName()` is used to load the JDBC driver. The developer must write routine that loads the JDBC / ODBC Bridge. The bridge driver called sun.jdbc.odbc.JdbcOdbcDriver. It is done in following way:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

Connect to the DBMS

After loading the driver the application must get connected to DBMS. For this we use DriverManager.getConnection() method. The DriverManager is highest class in Java.sql hierarchy and is responsible for managing driver related information.

The DriverManager.getConncetion() method is passed the URL of the database and user ID and password required by the database. The URL is the string object that contains the driver name that is being accessed by the Java program.

The DriverManager.getConncetion() method returns Connection interface that is used throughout the process to reference the database. The signature of this method is:

```
Connection DriverManager.getConncetion(String url,
                    String userID, String password);
```

Here, the URL format is specified as follows:

```
<protocol>:<subprotocol>:<dsn-name>
```

The 'protocol' is a JDBC protocol that is used to read the URL. The 'subprotocol' is JDBC driver name and 'dsn-name' is the name of the database that we provided while creating JDBC Bridge though control panel. We use the following URL for our application:

```
jdbc:odbc:customer
```

here, 'customer' is an example of DSN name given to our database. The user name and password are also provided at the time of creating DSN. It is not compulsory to provide the username and password. For example:

```
Conncetion con;
con = DriverManager.getConnection("jdbc:odbc:customer",
                                  "micro", "pitch");
```

Create Statement object

The createStatement( ) method of Connection interface is used to create the Statement object which is then used to execute the query. For example:

```
Statement st = con.createStatement();
```

Execute the query

The executeQuery() method of Statement object is used execute and process the query which returns the ResultSet object. ResultSet is the object which actually contains the result returned by the query. For example:

```
ResultSet rs = st.executeQuery("select * from customer");
```

Here, the 'customer' is neither database name nor DSN name but it is a table name.

Process the results

The ResultSet object is assigned the results received from the DBMS after the query is processed. The ResultSet object consists of methods used to interact with data that is returned by the DBMS to Java application program. For example, the next() method is used to proceed throughout the result set. It returns true, if the data is available in result set to read.

The ResultSet also contains several getXxx( ) methods to read the value from particular column of current row. For example, getString("name") will read the value from column 'name' in the form of string. Instead of passing column name as parameter, we can pass column as parameter also. Such as, getString(1). For example:

```
String name;
int age;
do
{
     name = rs.getString("name");
     age = rs.getInt("age");
     System.out.println(name+"="+age);
```

```
} while(rs.next());
```

Terminate the Connection

The Connection to the DBMS is terminated by using the close() method of the Connection object once Java program has finished accessing the DBMS. The close( ) method throws as exception if problem is encountered when disengaging the DBMS. For example:

```
con.close();
```

The close() method of Statement object is used to close the statement object to stop the further processing.

Statement Objects [Ref. 1]

Once the connection to the database is opened, the Java application creates and sends a query to access data contained in the database. One of three type of statement objects is used to execute the query immediatelt. A PreparedStatement is used to execute the compiled query and CallableStetement is used to execute the stored procedure.

Statement object

The Statement object is used whenever a Java program needs to immediately execute a query without first having query compiled. The Statement contains three different methods depending upon the type of query these will be used.

1. **executeQuery()**

   This method returns the ResultSet object that contains rows, columns and metadata that represent data requested by the query. Its signature is:

   ```
   ResultSet executeQuery(String query);
   ```

   Generally, this method is used to execute only the ‘SELECT’ query of the SQL.

2. **executeUpdate()**

   This method is used to execute the queries that contain INSERT, DELETE and UPDATE statements. This method returns integer indicating the number of rows that were updated by the query. Its signature is:

```
int executeUpdate(String query);
```

For example:

```
int rows = st.executeUpdate("DELETE FROM EMPLOYEES
                                    WHERE STATUS=0");
```

3. **execute()**

It executes the given SQL statement, which may return multiple results. In some (uncommon) situations, a single SQL statement may return multiple result sets and/or update counts we must then use the methods getResultSet( ) or getUpdateCount( ) to retrieve the result, and getMoreResults( ) to move to any subsequent result(s). Signature is as follows:

```
public boolean execute(String sql)
```

For example:

```
if(st.execute())
  rs = st.getResultSet();
```

Signatures of other methods:

```
public ResultSet getResultSet()
public int getUpdateCount()
public boolean getMoreResults()
```

PreparedStatement object

A SQL query must be compiled before the DBMS processes the query. Compiling occurs after one of the Statement object's execution method is called. Compiling a query is an overhead that is acceptable if the query is called once. However, compiling process can become an expensive overhead if the query is executed several times by the same program during the same session.

A SQL query can be precompiled and executed by using the PreparedStatement object. In such cases a query is created similar to other queries. However, a question mark is given on the place for the value that is inserted into the query after it is compiled. It is the value that changes each time the query is executed.

For doing this process, we need to construct the query with question marks such as,

```
"select * from nation where population > ?"
```

Such type of the query is passed as the parameter to the prepareStatement( ) method of the Connection object which then returns the PreparedStatement object. For example:

```
String query = “select * from nation where population > ?”;
PreparedStatement ps = prepareStatement(query);
```

Once the PreparedStatement object is obtained, the setXxx( ) methods of it can be used to replace question mark with the value passed to setXxx() method. There are a number of setXxx() methods available in PreparedStatement object, each of which specifies the data type of value that is being passed to setXxx() method. For example, considering the above query again,

```
ps.setInt(1, 100000);
```

This method requires two parameters. First parameter is an integer that identifies position of the question mark placeholder and second is the value that replaces the question mark. If the query contains two question marks we have to pass second value also using setXxx() method.

Now, we need to use appropriate execute method depending upon type of the query without any parameters. Such as,

```
ResultSet rs = ps.executeQuery();
```

This will generate the ResultSet object as the execution of the query. The PreparedStatement contain all three execute methods but without any parameters as given below:

```
ResultSet executeQuery( )
int executeUpdate( )
boolean execute( )
```

The setXxx( ) methods:

```
void setBoolean(int index, boolean value);
void setByte(int index, byte value);
void setDate(int index, Date value);
void setDouble(int index, double value);
void setFloat(int index, float value);
void setInt(int index, int value);
void setLong(int index, long value);
void setObject(int index, Object value);
void setShort(int index, short value);
void setString(int index, String value);
```

Example:
Consider the following database:

	Roll	Name	Marks
	1	Rakhee	75
	2	Amit	49
	3	Ajita	63
	4	Rahul	78
	5	Minal	67
	6	Karthik	71

```
import java.sql.*;
class StudentData
{
   public static void main(String args[])
   {
      try
      {
       Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       Connection con =
                  DriverManager.getConnection("jdbc:odbc:stud");
       PreparedStatement ps = con.prepareStatement("select *
                  from Student where Marks > ?");
       ps.setInt(1,70); //set question marks place holder
       ResultSet rs = ps.executeQuery(); //execute
       System.out.println("Students having marks > 70 are:");
       while(rs.next())
           System.out.println(rs.getString(2));
       con.close();
      }
      catch(Exception e){ }
   }
}
```

Output:

```
Students having marks > 70 are:
Rakhee
Rahul
Karthik
```

CallableStatement Object

The CallableStatement is used to call the stored procedures from within a JDBC application program. A stored procedure is a block of code and is identified by a unique name. The type style of code depends upon the DBMS vendor and can be written in PL/SQL, Transact-SQL, C or another programming language. The stored procedure is executed by invoking name of the stored procedure. For example, a stored procedure written in PL/SQL as given below:

```
CREATE PROCEDURE sp_interest
(id IN INTEGER,
bal IN OUT FLOAT) IS
BEGIN
SELECT balance
INTO bal
FROM account
WHERE account_id = id;

bal := bal + bal * 0.03;

UPDATE account
SET balance = bal
WHERE account_id = id;

END;
```

The CallableStatement object uses three types of parameters when calling a stored procedure. These parameters are IN, OUT, INOUT. The IN parameter contains the data that needs to be passed to the stored procedure whose value is assigned using setXxx() method. Whereas, OUT parameter contains the value returned by the stored procedure, if any. The OUT parameter must be registered using registerOutParameter() method and afterwards this is retrieved by using getXxx() method. The INOUT parameter is a single parameter that is used to both pass information and retrieve information from a stored procedure.

Consider above example, the name of the stored procedure is given as, sp_interest. Its definition is very similar to those of Java method definition. The variable 'id' is an input integer parameter passed to this procedure and variable 'bal' is the float parameter acting as input and output both. The stored procedure contains the SQL query code to perform certain operations depending upon input value to the stored procedure and it returns the value in variable 'bal'. We can now write our code to call this procedure to pass the parameters and to retrieve the information.

After establishing the connection, the prepareCall() method of the Connection object is passed with query of stored procedure call. It returns the object of CallableStatement. The OUT parameter of the procedure must be registered using registerOutParemeter() method which contains following general form:

```
public void registerOutParameter(int parameterIndex, int sqlType)
```

here, *parameterIndex* refers to the index of the parameter passed to that stored procedure. And *sqlType* is type of the value which is expected to retrieve from stored procedure. Generally, *sqlType* is the value of type java.sql.Types.

The execute() method of the CallableStatement object is called next to execute the query. This method does not require any query because the query is already identified by prepareCall() method of CallableStatement. After the stored procedure is executed, the getString() method is called to return the value of specified parameter of the stored procedure. See the example given below:

```
try
{
    CallableStatement statement;
     statement = c.prepareCall("{call sp_interest(?,?)}");
     statement.registerOutParameter(2, java.sql.Types.FLOAT);
    statement.setInt(1, 310);
    statement.execute( );
    System.out.println("New balance:" + statement.getFloat(2));
}
statement.close( );
c.close( );
```

ResultSet [Ref. 1]

The ResultSet class provides methods to access data generated by a table query. This includes a series of get methods which retrieve data in any one of the JDBC SQL type formats, either by column number or by column name.

The ResultSet object contains methods that are used to copy data from the ResultSet into Java collection object or variable for further processing. Data in ResultSet object is logically organized into virtual table consisting of rows and columns. In addition to the data, the ResultSet object also contains metadata such as column names, column size, and column data type.

The ResultSet uses a virtual cursor to point to the row of virtual table as shown in figure below:

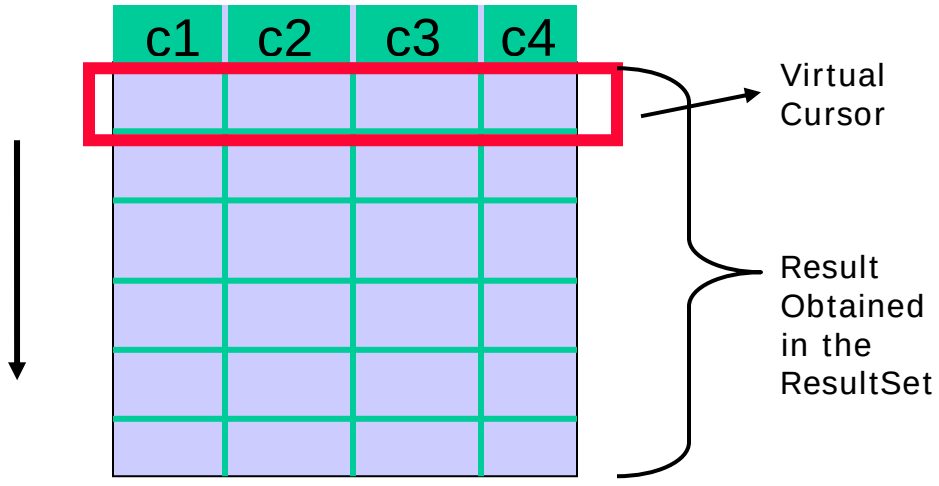


Fig. ResultSet operations

A JDBC application program must move the virtual cursor to each row and then move the virtual cursor to each row and then use the methods of the ResultSet object to interact with the data stored in the column of that row. The virtual cursor is positioned above the first row of data when the ResultSet is returned by the executeQuery() method. This means that the virtual cursor must be moved to the first row using next() method. The next() method returns a boolean value if the row contains data otherwise false is returned indicating that no more rows exists in the ResultSet.

Once a virtual cursor points to a row the getXxx() method is used to copy the data from the row to a collection, object or a variable. The selection os particular get method depends upon the type of value that column contains. Following get methods are used in ResultSet:

```
boolean getBoolean(int columnIndex)
boolean getBoolean(String columnName)
byte getByte(int columnIndex)
byte getByte(String columnName)
Date getDate(int columnIndex)
Date getDate(String columnName)
double getDouble(int columnIndex)
double getDouble(String columnName)
float getFloat(int columnIndex)
float getFloat(String columnName)
int getInt(int columnIndex)
int getInt(String columnName)
long getLong(int columnIndex)
long getLong(String columnName)
double getDouble(int columnIndex)
double getDouble(String columnName)
Object getObject(int columnIndex)
Object getObject(String columnName)
short getShort(int columnIndex)
short getShort(String columnName)
String getString(int columnIndex)
String getString(String columnName)
Time getTime(int columnIndex)
Time getTime(String columnName)
```

Now, consider the following database,

	Roll	Name	Marks	Passed	Birthdate
	1	Rakhee	75	☑	12/14/1990
	2	Amit	38	☐	10/2/1990
	3	Ajita	63	☑	1/24/1989
	4	Rahul	78	☑	1/1/1990
	5	Minal	67	☑	2/12/1991
✎	6	Karthik	71	☑	7/6/1988

The program given below reads the database and displays its contents in a tabular format:

```
import java.sql.*;
 class StudentData
 {
    public static void main(String args[])
    {
       try
       {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection con =
               DriverManager.getConnection("jdbc:odbc:stud");
        Statement st = con.createStatement();
        ResultSet rs = st.executeQuery("select * from Student");
        System.out.println("The Database is:-");
        System.out.println("Roll\tName\t\tMarks Pass Birth-Date");
      System.out.println("====================================");
        while(rs.next())
        {
            int roll = rs.getInt(1);
            String name = rs.getString(2);
            int marks = rs.getInt("Marks");
            boolean pass = rs.getBoolean(4);
            Date d = rs.getDate(5);
            System.out.printf("%-5d",roll);
            System.out.printf("%-10s",name);
            System.out.printf("%-6d",marks);
            if(pass)
               System.out.printf("Yes  ");
            else
               System.out.printf("No   ");
            System.out.printf("%-15s\n",d.toString());
        }
        con.close();
       }
       catch(Exception e){ }
    }
 }
```

Output:

```
The Database is:-
Roll Name      Marks Pass Birth-Date
====================================
1    Rakhee    75    Yes  1990-12-14
2    Amit      38    No   1990-10-02
```

```
3    Ajita     63    Yes  1989-01-24
4    Rahul     78    Yes  1990-01-01
5    Minal     67    Yes  1991-02-12
6    Karthik   71    Yes  1988-07-06
```

Scrollable ResultSet

The addition to the next() method, the ResultSet can also be moved backward or any other position inside it. For doing these scrolling, we can use the following methods:

```
public boolean first()
public boolean last()
public boolean next()
public boolean previous()
public boolean absolute(int position)
public boolean relative(int rows)
```

The first() method moves the virtual cursor to the first row of the ResultSet. Likewise, last() method moves the virtual cursor to the last row of the ResultSet. The previous() method moves the virtual cursor to the previous row of the ResultSet from current position. The absolute() method positions virtual cursor at the row number specified by integer passed as parameter to this method. The relative() method moves the virtual cursor the specified number of rows virtual cursor. If this parameter is positive, virtual cursor moves forward by that number of rows. Similarly, negative number moves the cursor backward direction by that number of rows.

The Statement object that is created using createStatement() of Connection object must be set up to handle a scrollable ResultSet by passing createStatement method one of three constants given below:

```
TYPE_FORWARD_ONLY
TYPE_SCROLL_INSENSITIVE
TYPE_SCROLL_SENSITIVE
```

The TYPE_FORWARD_ONLY constant restricts the virtual cursor to downward movement, which is a default setting. The TYPE_SCROLL_INSENSITIVE and TYPE_SCROLL_SENSITIVE constants permit virtual cursor to move in both directions. The TYPE_SCROLL_INSENSITIVE constant makes the ResultSet insensitive to make to changes made by another JDBC application to data in the table whose rows are reflected in the ResultSet. The TYPE_SCROLL_SENSITIVE constant makes the ResultSet sensitive to those changes.

Updatable ResultSet

The rows contained in the ResultSet can be updatable similar to how rows in the current table can be updated. This is made possible by passing the createStatement() method of the Connection object the CONCUR_UPDATABLE. Alternatively, the CONCUR_READ_ONLY constant can be passed to the createStatement() method to prevent the ResultSet from being updated.

There are three ways in which the ResultSet can be changed. These are updating values in a row and inserting a new row. All these changes are accomplished by using methods of Statement object.

Updating ResultSet

Once the executeQuery() method of the Statement object returns the ResultSet, the updateXxx() method is used to change the value of column in the current row of the ResultSet. The xxx is replaced by any data type of the column that is to be updated.

The updateXxx() method requires two parameters. The first is either the number or the name of the ResultSet that is being updated and the second parameter is the value in the column of the ResultSet. Such as,

```
void updateInt(int columnIndex, int x)
void updateInt(String columnName, int x)
void updateLong(int columnIndex, long x)
void updateLong(String columnName, long x)
```

and so on….

For example:

```
s.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                        ResultSet.CONCUR_UPDATABLE);
r.updateString(“Name”, “Alisha”);
r.updateRow();
```

In above code, the current row’s column ‘Name’s value will be replaced by ‘Alisha’.

Deleting a Row

The deleteRow() method is used to remove a row from ResultSet. This method directly deletes the contents on the current row and the virtual cursor will be moved to the next row automatically. For example:

```
rs.deleteRow();
```

Inserting a Row

Inserting a row in the ResultSet is accomplished using basically the same technique as is used to update the ResultSet. That is, the updateXxx() method is used to specify the column and the value that will be placed into the column of the ResultSet.

The updateXxx() requires two parameters. The first is either name of the column or number of the column of the ResultSet. The second parameter is new value that will be placed in the column of the ResultSet. The insertRow() method is called after the updateXxx() methods, which causes a new row to be inserted into the ResultSet having values that reflect the parameters in the updateXxx() methods. This also updates underlying database. For example:

```
s.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_UPDATABLE );
r.updateString(“Name”, “Aneeta”);
r.updateInt(“Roll”, 12);
r.insertRow();
```

Metadata [Ref. 1]

Metadata is nothing but data about data. Database name, table name, column name, column attributes, and other information that describes database components are known as Metadata. For Example, the size of client’s first name column describes the data contained within the column and therefore is referred to as metadata.

Metadata is used by JDBC application to identify database components without needing to know details of a column, the table or the database. For example, a J2EE component can request from the DBMS the data type of a specific column. The column type is used by a JDBC application to copy data retrieved from the DBMS into a Java collection.

JDBC provides two meta-data interfaces:

- `java.sql.ResultSetMetaData`
- `java.sql.DatabaseMetaData.`

The meta-data described by these classes was included in the original JDBC ResultSet and Connection classes. The extra functionality could be served by creating meta-data classes to provide the often esoteric information required by a minority of developers.

DatabaseMetaData

This interface provides methods that tell us about the database for a given Connection object, including:

- What tables exist in the database visible to the user?

- What username is being used by this connection?
- Is this database connection read-only?
- What keywords are used by the database, that are not SQL2?
- Does the database support column aliasing?
- Are multiple result sets from a single execute() call supported?
- Are outer joins supported?
- What are the primary keys for a table?

The JDBC application program retrieves the object of DatabaseMetaData by using the getMetaData of the Connection object. Such as,

```
DatabaseMetaData dm = con.getMetaData();
```

Then we can use the following methods of DatabaseMetaData interface to retrieve the metadata from the database.

1. **getDatabaseProductName()**
   It returns product name of database

2. **getDatabaseProductVersion()**
   It returns product version of database

3. **getUserName()**
   It returns the user name

4. **getURL()**
   It returns URL of database

5. **getSchemas()**
   It returns schema names available in database

6. **getPrimaryKeys()**
   It returns primary keys

7. **getProcedures()**
   It returns stored procedure names

8. **getTables()**
   It returns names of tables in database

ResultSetMetaData:

The ResultSetMetaData class provides extra information about ResultSet objects returned from a database query. This class provides us with answers to the following questions:

- How many columns are in the result set?
- Are column names case-sensitive?
- Can we search on a given column?
- Is NULL a valid value for a given column?
- How many characters is the maximum display size for a given column?
- What label should be used in a display header for the column?
- What is the name of a given column?
- What table did a given column come from?
- What is the data-type of a given column?

Metadata that describes the ResultSet is retrieved by calling the getMetaData() method of ResultSet object. This returns a ResultSetMetaData object as,

```
ResultSetMetaData rm = Result.getMetaData()
```

Once the ResultSetMetaData is retrieved, the JDBC application can call methods of ResulSetMetaData object to retrieve specific kind of metadata. The more commonly called methods are as follows:

1. **getColumnCount()**
   It returns no. of columns contained in ResultSet.

2. **getColumnName(int number)**
   It returns the name of column specified by column number.

3. **getColumnType(int no)**
   It returns the data type of column specified by column number.

For example:

```
ResultSetMetaData md = rs.getMetaData();
// get number of columns
int nCols = md.getColumnCount();
// print column names
for(int i=1; i < nCols; ++i)
     System.out.print(md.getColumnName(i)+",");
// output resultset
while (rs.next())
{    for(int i=1; i < nCols; ++i)
          System.out.print(rs.getString(i)+",");
     System.out.println(rs.getString(nCols));
}
```

Compatibility of JDBC Data Types and Java Data Types

JDBC Type	Java Type
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	
BINARY	byte[]
VARBINARY	
LONGVARBINARY	
CHAR	String
VARCHAR	
LONGVARCHAR	
NUMERIC	BigDecimal
DECIMAL	
DATE	java.sql.Date
TIME	java.sql.Timestamp
TIMESTAMP	
CLOB	Clob*
BLOB	Blob*
ARRAY	Array*
DISTINCT	mapping of underlying type
STRUCT	Struct*
REF	Ref*
JAVA_OBJECT	underlying Java class

* Objects of java.sql package

SQL Exceptions [Ref. 1]

There are three kinds of exceptions that are thrown by JDBC methods. These are:

- SQLException
- SQLWarnings
- DataTruncation

SQLException

It commonly reflects the SQL syntax error in the query and is thrown many of the methods contained in java.sql package.

This exception is most commonly caused by connectivity issues with the database. It can also caused by subtle coding errors like trying to access an

object that is been closed. Following SQLException class specific methods listed below:

1. **getMessage()**
   It gives the description of the error.
2. **getSQLState()**
   It returns SQLState (Open Group SQL specification) identifying the exception.
3. **getErrorCode()**
   It returns a vendor-specific integer error code.
4. **getNextException()**
   If more than one error occurs, they are chained together. This method is used chain to the next exception.

For example:

```
try
{
     ... // JDBC statement
} catch (SQLException sqle) {
  while (sqle != null) {
    System.out.println("Message:  " + sqle.getMessage());
    System.out.println("SQLState: " + sqle.getSQLState());
    System.out.println("Vendor Error: " +
                       sqle.getErrorCode());
    sqle.printStackTrace(System.out);
    sqle = sqle.getNextException();
  }
}
```

SQLWarning

SQLWarnings are rare, but provide information about the database access warnings. These are chained to object whose method produced the warning. The following objects such as Connection, Statement (also, PreparedStatement, CallableStatement) ResultSet can receive a warning. We can use getWarning() method to obtain the warning object, and getNextWarning (on the warning object) for any additional warnings. These warnings are cleared on the object each time the statement is executed.
For example:

```
ResultSet results = statement.executeQuery(someQuery);
SQLWarning warning = statement.getWarnings();
while (warning != null) {
  System.out.println("Message: " + warning.getMessage());
  System.out.println("SQLState: " + warning.getSQLState());
  System.out.println("Vendor Error: " +
                     warning.getErrorCode());
  warning = warning.getNextWarning();
```

```
}
while (results.next()) {
  int value = rs.getInt(1);
     ... // Call additonal methods on result set.
  SQLWarning warning = results.getWarnings();
  while (warning != null) {
    System.out.println("Message: " + warning.getMessage());
    System.out.println("SQLState: " + warning.getSQLState());
    System.out.println("Vendor Error: " +
                       warning.getErrorCode());
    warning = warning.getNextWarning();
  }
 }
```

Whenever the data is lost due to truncation of the data value, the DataTruncation exception is thrown.

References

1. **J2EE the Complete Reference**,
   First Edition by Jim Keogh, 2002 Tata McGraw Hill,
   Chapter 5: J2EE Database Concepts
   Chapter 6: JDBC Objects
   Chapter 7: JDBC and Embedded SQL
   (Most of the data is referred from this book)

2. **Mastering Java 2,**
   First Indian Edition by John Zukowski, 2000, BPB Publications
   Chapter 21: Java Database Connectivity

3. **Database Programming with JDBC and Java,**
   First Edition by Pratik Patel, 1996, Coriolis, The Coriolis Group

4. **Java Database Programming with JDBC,**
   Second Edition by George Reese, 2001, O'Reilly & Associates,

5. **JDBC: Java Database Connectivity,**
   First Edition by Bernard Van Haecke, 1997, IDG Books Worldwide Inc.,

6. **JDBC™ API Tutorial and Reference,**
   Third Edition by Maydene Fisher, Jon Ellis, Jonathan Bruce, 2003, Addison Wesley (This book is recognized by Sun Microsystems Inc.)

7. **JDK 5.0 Documentation,**
   Sun Microsystems Inc., USA, www.java.sun.com

Chapter 04
The Tour of Swing

Lectures allotted: 08
Marks Given: 14

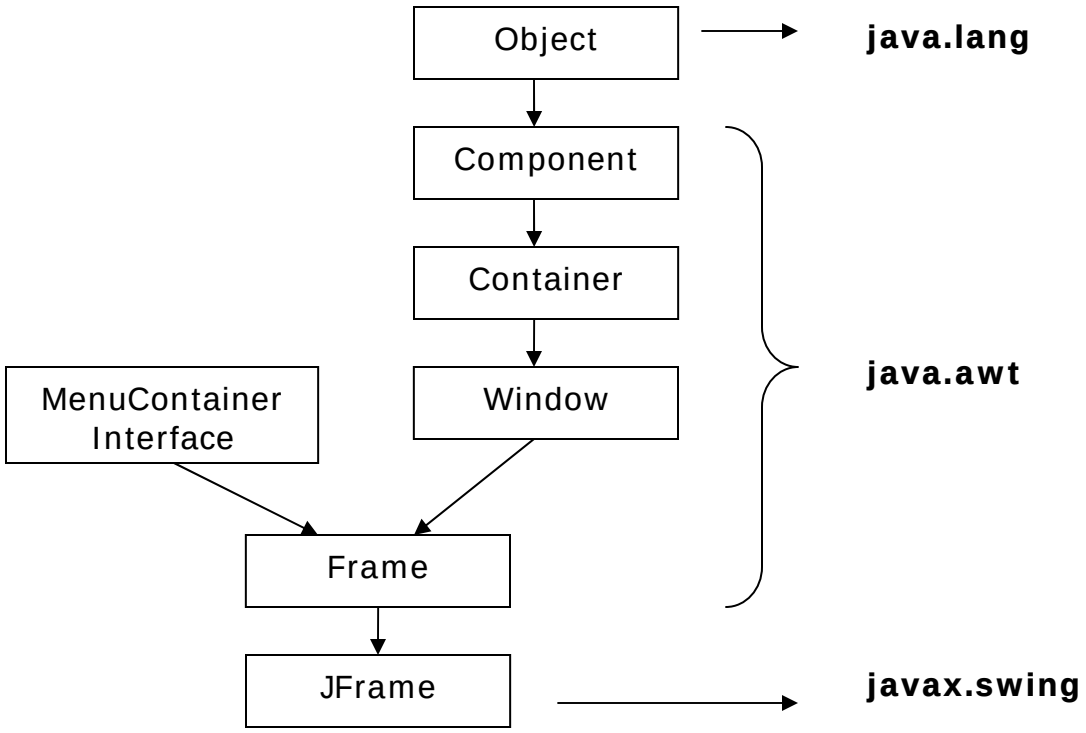
Contents:

4.2 Japplet
Icons and Labels
Text Fields
Buttons
Combo Boxes
Checkboxes
Tabbed Panes
Scroll Panes
4.3 Trees
Tables
Exploring the Swings

Introduction

Swing is a set of classes this provides more powerful and flexible components than are possible with the AWT. In addition to the familiar components, such as buttons, check boxes, and labels, Swing supplies several exciting additions, including tabbed panes, scroll panes, trees, and tables. Even familiar components such as buttons have more capabilities in Swing. For example, a button may have both an image and a text string associated with it. Also, the image can be changed as the state of the button changes.

Unlike AWT components, Swing components are not implemented by platform-specific code. Instead, they are written entirely in Java and, therefore, are platform-independent. The term lightweight is used to describe such elements. The number of classes and interfaces in the Swing packages is substantial. Swing is the set of packages built on top of the AWT that provide us with a great number of pre-built classes that is, over 250 classes and 40 UI components.



The Swing component classes that are shown below:

Class	Description
AbstractButton	Abstract super-class for Swing buttons.
ButtonGroup	Encapsulates a mutually exclusive set of buttons.
ImageIcon	Encapsulates an icon.
JApplet	The Swing version of Applet.
JButton	The Swing push button class.
JCheckBox	The Swing check box class.

JComboBox	Encapsulates a combo box (a combination of a drop-down list and text field).
JLabel	The Swing version of a label.
JRadioButton	The Swing version of a radio button.
JScrollPane	Encapsulates a scrollable window.
JTabbedPane	Encapsulates a tabbed window.
JTable	Encapsulates a table-based control.
JTextField	The Swing version of a text field.
JTree	Encapsulates a tree-based control.

The Swing-related classes are contained in javax.swing and its subpackages, such as javax.swing.tree.

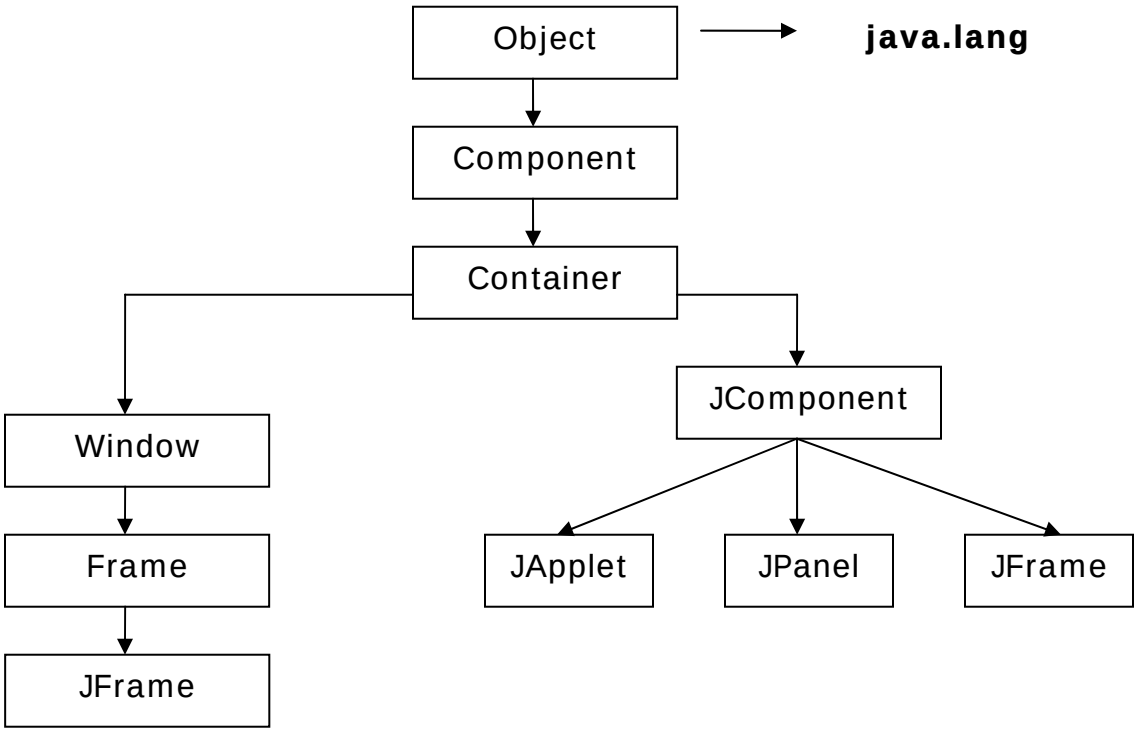


Fig. The Swing family tree (Ref. No. 2)

Swing Features (Ref. No. 2)

Besides the large array of components in Swing and the fact that they are lightweight, Swing introduces many other innovations.

- **Borders**

  We can draw borders in many different styles around components using the setborder( ) method.

- **Graphics Debugging**

We can use setDebuggingGraphicsOptions method to set up graphics debugging which means among the other things, that you can watch each line as its drawn and make it flash.

- **Easy mouseless operation**
  It is easy to connect keystrokes to components.

- **Tooltips**
  We can use the setToolTipText method of JComponent to give components a tooltip, one of those small windows that appear when the mouse hovers over a component and gives explanatory text.

- **Easy Scrolling**
  We can connect scrolling to various components-something that was impossible in AWT.

- **Pluggable look and feel**
  We can set the appearance of applets and applications to one of three standard looks. Windows, Motif (Unix) or Metal (Standard swing look).

- **New Layout Managers**
  Swing introduces the BoxLayout and OverlayLayout layout managers.

One of the differences between the AWT and Swing is that, when we redraw items on the screen of AWT, the update method is called first to redraw the item's background and programmers often override update method to just call the paint method directly to avoid flickering. In Swing, on the other hand the update method does not redraw the item's background because components can be transparent; instead update just calls paint method directly.

JApplet

Fundamental to Swing is the JApplet class, which extends Applet. Applets that use Swing must be subclasses of JApplet. JApplet is rich with functionality that is not found in Applet. For example, JApplet supports various "panes," such as the content pane, the glass pane, and the root pane.

When adding a component to an instance of JApplet, do not invoke the add( ) method of the applet. Instead, call add( ) for the content pane of the JApplet object. The content pane can be obtained via the method shown here:

```
Container getContentPane( )
```

The add( ) method of Container can be used to add a component to a content pane. Its form is shown here:

```
void add(comp)
```

Here, *comp* is the component to be added to the content pane.

Icons and Labels

In Swing, icons are encapsulated by the ImageIcon class, which paints an icon from an image. Two of its constructors are shown here:

```
ImageIcon(String filename)
ImageIcon(URL url)
```

The first form uses the image in the file named *filename*. The second form uses the image in the resource identified by *url*. The ImageIcon class implements the Icon interface that declares the methods shown here:

Method	**Description**
int getIconHeight( )	Returns the height of the icon in pixels.
int getIconWidth( )	Returns the width of the icon in pixels.
void paintIcon(Component comp, Graphics g, int x, int y)	Paints the icon at position x,y on the graphics context g. Additional information about the paint operation can be provided in comp.

Swing labels are instances of the JLabel class, which extends JComponent. It can display text and/or an icon. Some of its constructors are shown here:

```
JLabel(Icon i)
Label(String s)
JLabel(String s, Icon i, int align)
```

Here, *s* and *i* are the text and icon used for the label. The *align* argument is either LEFT, RIGHT,CENTER, LEADING, or TRAILING. These constants are defined in the SwingConstants interface, along with several others used by the Swing classes. The icon and text associated with the label can be read and written by the following methods:

```
Icon getIcon( )
String getText( )
void setIcon(Icon i)
void setText(String s)
```

Here, *i* and *s* are the icon and text, respectively. The following example illustrates how to create and display a label containing both an icon and a string. The applet begins by getting its content pane. Next, an ImageIcon object is created for the file IC.jpg. This is used as the second argument to the JLabel constructor. The first and last arguments for the JLabel constructor are the label text and the alignment. Finally, the label is added to the content pane.

```
import java.awt.*;
import javax.swing.*;
/* <applet code="JLabelDemo" width=250 height=150> </applet> */
public class JLabelDemo extends JApplet
{
        public void init()
        {
                Container contentPane = getContentPane();
                ImageIcon ii = new ImageIcon("IC.jpg");
                JLabel jl = new JLabel("IC", ii, JLabel.CENTER);
                contentPane.add(jl);
        }
}
```

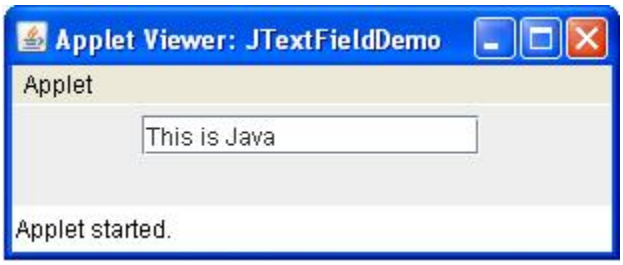
Text Fields

The Swing text field is encapsulated by the JTextComponent class, which extends JComponent. It provides functionality that is common to Swing text components. One of its subclasses is JTextField, which allows us to edit one line of text. Some of its constructors are shown here:

```
JTextField( )
JTextField(int cols)
JTextField(String s, int cols)
JTextField(String s)
```

Here, *s* is the string to be presented, and *cols* is the number of columns in the text field. The following example illustrates how to create a text field. The applet begins by getting its content pane, and then a flow layout is assigned as its layout manager. Next, a JTextField object is created and is added to the content pane.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300 height=50>
</applet>
*/
public class JTextFieldDemo extends JApplet
{
    JTextField jtf;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        jtf = new JTextField(15);
        contentPane.add(jtf);
    }
}
```



Buttons

Swing buttons provide features that are not found in the Button class defined by the AWT. For example, we can associate an icon with a Swing button. Swing buttons are subclasses of the AbstractButton class, which extends JComponent. AbstractButton contains many methods that allow us to control the behavior of buttons, check box and radio buttons. For example, we can define different icons that are displayed for the component when it is disabled, pressed, or selected. Another icon can be used as rollover icon, which is displayed when the mouse is positioned over that component. The following are the methods that control this behavior:

```
void setDisabledIcon(Icon di)
void setPressedIcon(Icon pi)
void setSelectedIcon(Icon si)
```

```
void setRolloverIcon(Icon ri)
```

Here, *di, pi, si,* and *ri* are the icons to be used for these different conditions. The text associated with a button can be read and written via the following methods:

```
String getText( )
void setText(String s)
```

Here, *s* is the text to be associated with the button.

Concrete subclasses of AbstractButton generate action events when they are pressed. Listeners register and un-register for these events via the methods shown here:

```
void addActionListener(ActionListener al)
void removeActionListener(ActionListener al)
```

Here, *al* is the action listener. AbstractButton is a superclass for push buttons, check boxes, and radio buttons.

JButton Class

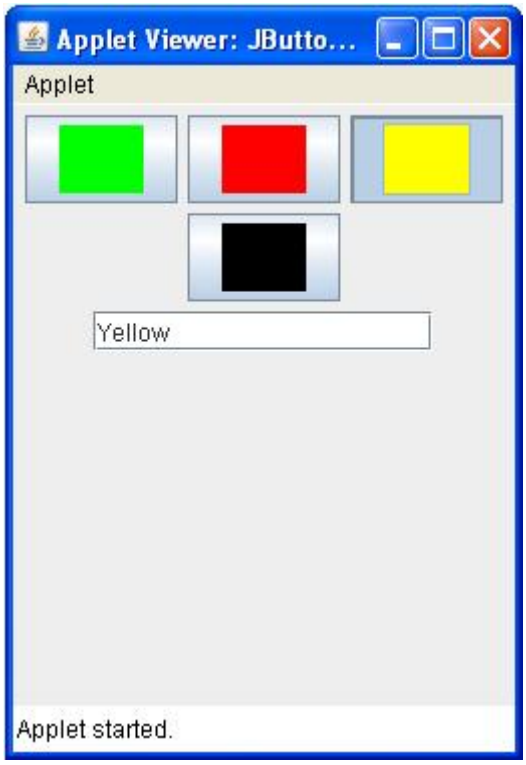
The JButton class provides the functionality of a push button. JButton allows an icon string, or both to be associated with the push button. Some of its constructors are shown here:

```
JButton(Icon i)
JButton(String s)
JButton(String s, Icon i)
```

Here, *s* and *i* are the string and icon used for the button. The following example displays four push buttons and a text field. Each button displays an icon that represents the flag of a country. When a button is pressed, the name of that country is displayed in the text field. The applet begins by getting its content pane and setting the layout manager of that pane. Four image buttons are created and added to the content pane. Next, the applet is registered to receive action events that are generated by the buttons. A text field is then created and added to the applet. Finally, a handler for action events displays the command string that is associated with the button. The text field is used to present this string.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JButtonDemo" width=250 height=300>
```

```
</applet>
*/
public class JButtonDemo extends JApplet
implements ActionListener
{
    JTextField jtf;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        ImageIcon france = new ImageIcon("green.jpg");
        JButton jb = new JButton(france);
        jb.setActionCommand("Green");
        jb.addActionListener(this);
        contentPane.add(jb);
        ImageIcon germany = new ImageIcon("red.jpg");
        jb = new JButton(germany);
        jb.setActionCommand("Red");
        jb.addActionListener(this);
        contentPane.add(jb);
        ImageIcon italy = new ImageIcon("yellow.jpg");
        jb = new JButton(italy);
        jb.setActionCommand("Yellow");
        jb.addActionListener(this);
        contentPane.add(jb);
        ImageIcon japan = new ImageIcon("black.jpg");
        jb = new JButton(japan);
        jb.setActionCommand("Black");
        jb.addActionListener(this);
        contentPane.add(jb);
        jtf = new JTextField(15);
        contentPane.add(jtf);
    }
    public void actionPerformed(ActionEvent ae)
    {
        jtf.setText(ae.getActionCommand());
    }
}
```



Check Boxes

The **JCheckBox** class, which provides the functionality of a check box, is a concrete implementation of AbstractButton. **It is immediate super-class is JToggleButton, which provides support for two-state buttons**. Some of its constructors are shown here:

```
JCheckBox(Icon i)
JCheckBox(Icon i, boolean state)
JCheckBox(String s)
JCheckBox(String s, boolean state)
JCheckBox(String s, Icon i)
JCheckBox(String s, Icon i, boolean state)
```

Here, *i* is the icon for the button. The text is specified by *s*. If *state* is true, the check box is initially selected. Otherwise, it is not. The state of the check box can be changed via the following method:

```
void setSelected(boolean state)
```

Here, *state* is true if the check box should be checked. The following example illustrates how to create an applet that displays four check boxes and a text field. When a check box is pressed, its text is displayed in the text field the content pane for the JApplet object is obtained, and a flow layout is assigned as its layout manager. Next, four check boxes are added to the

content pane, and icons are assigned for the normal, rollover, and selected states. The applet is then registered to receive item events. Finally, a text field is added to the content pane. When a check box is selected or deselected, an item event is generated. This is handled by itemStateChanged( ). Inside itemStateChanged( ), the getItem( ) method gets the JCheckBox object that generated the event. The getText( ) method gets the tex for that check box and uses it to set the text inside the text field.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo" width=400 height=50>
</applet>
*/
public class JCheckBoxDemo extends JApplet
implements ItemListener
{
    JTextField jtf;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        JCheckBox cb = new JCheckBox("C", true);
        cb.addItemListener(this);
        contentPane.add(cb);
        cb = new JCheckBox("C++", false);
        cb.addItemListener(this);
        contentPane.add(cb);
        cb = new JCheckBox("Java", false);
        cb.addItemListener(this);
        contentPane.add(cb);
        cb = new JCheckBox("Perl", false);
        cb.addItemListener(this);
        contentPane.add(cb);
        jtf = new JTextField(15);
        contentPane.add(jtf);
    }
    public void itemStateChanged(ItemEvent ie)
    {
        JCheckBox cb = (JCheckBox)ie.getItem();
        jtf.setText(cb.getText());
    }
}
```

Radio Buttons

Radio buttons are supported by the JRadioButton class, which is a concrete implementation of AbstractButton. Its immediate super-class is JToggleButton, which provides support for two-state buttons. Some of its constructors are shown here:

```
JRadioButton(Icon i)
JRadioButton(Icon i, boolean state)
JRadioButton(String s)
JRadioButton(String s, boolean state)
JRadioButton(String s, Icon i)
JRadioButton(String s, Icon i, boolean state)
```

Here, *i* is the icon for the button. The text is specified by s. If *state* istrue, the button is initially selected. Otherwise, it is not. Radio buttons must be configured into a group. Only one of the buttons in that group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected.

The ButtonGroup class is instantiated to create a button group. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:

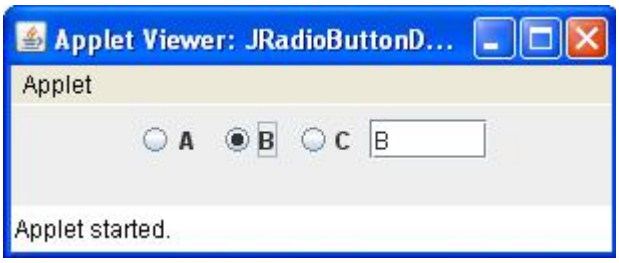
```
void add(AbstractButton ab)
```

Here, *ab* is a reference to the button to be added to the group. The following example illustrates how to use radio buttons. Three radio buttons and one text field are created. When a radio button is pressed, its text is displayed in the text field. First, the content pane for the JApplet object is obtained and a flow layout is assigned as its layout manager. Next, three radio buttons are added to the content pane. Then, a button group is defined and the buttons are added to it. Finally, a text field is added to the content pane.

Radio button presses generate action events that are handled by actionPerformed(). The getActionCommand() method gets the text that is associated with a radio button and uses it to set the text field.

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

```
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JRadioButtonDemo" width=300 height=50>
</applet>
*/
public class JRadioButtonDemo extends JApplet
implements ActionListener
{
    JTextField tf;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());

        JRadioButton b1 = new JRadioButton("A");
        b1.addActionListener(this);
        contentPane.add(b1);

        JRadioButton b2 = new JRadioButton("B");
        b2.addActionListener(this);
        contentPane.add(b2);

        JRadioButton b3 = new JRadioButton("C");
        b3.addActionListener(this);
        contentPane.add(b3);

        ButtonGroup bg = new ButtonGroup();
        bg.add(b1);
        bg.add(b2);
        bg.add(b3);
        tf = new JTextField(5);
        contentPane.add(tf);
    }
    public void actionPerformed(ActionEvent ae)
    {
        tf.setText(ae.getActionCommand());
    }
}
```



Combo Boxes

Swing provides a combo box (a combination of a text field and a drop-down list) through the JComboBox class, which extends JComponent. A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry. We can also type our selection into the text field. Two of JComboBox's constructors are shown here:

```
JComboBox( )
JComboBox(Vector v)
JComboBox(Objects obj[])
```

Here, *v* is a vector that initializes the combo box and *obj* is the array of objects. Items are added to the list of choices via the addItem( ) method, whose signature is shown here:

```
void addItem(Object obj)
```

Here, *obj* is the object to be added to the combo box.

Important Methods:

`public void setEditable(boolean aFlag)`
It determines whether the JComboBox field is editable or not?

`public boolean isEditable()`
It returns true if the JComboBox is editable. By default, a combo box is not editable.

`public void setMaximumRowCount(int count)`
It sets the maximum number of rows the JComboBox displays. If the number of objects in the model is greater than 'count', the combo box uses a scrollbar.

`public void setSelectedItem(Object anObject)`
It sets the selected item in the combo box display area to the object in the argument. If anObject is in the list, the display area shows anObject selected.

`public void insertItemAt(Object anObject, int index)`
It inserts an item 'anObject' into the item list at a given 'index'.

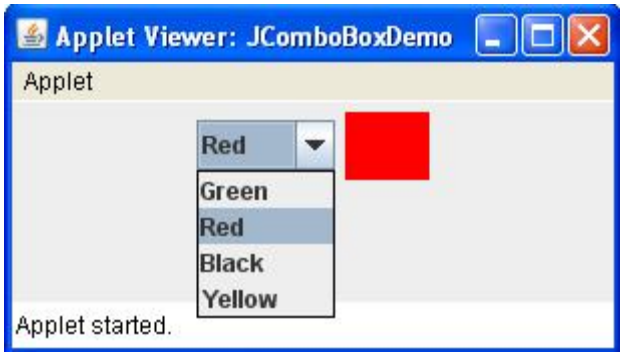
`public void removeItem(Object anObject)`
It removes an item 'anObject' from the item list.

`public void removeItemAt(int anIndex)`

It removes the item at ‘anIndex’.

The following example contains a combo box and a label. The label displays an icon. The combo box contains entries for colors Green, Red, Yellow and Black. When a country is selected, the label is updated to display the color for that particular color. Color jpeg images are already stored in the current directory.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JComboBoxDemo" width=300 height=100>
</applet>
*/
public class JComboBoxDemo extends JApplet
implements ItemListener
{
    JLabel jl;
    ImageIcon green, red, black, yellow;
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new FlowLayout());
        JComboBox jc = new JComboBox();
        jc.addItem("Green");
        jc.addItem("Red");
        jc.addItem("Black");
        jc.addItem("Yellow");
        jc.addItemListener(this);
        contentPane.add(jc);
        jl = new JLabel(new ImageIcon("green.jpg"));
        contentPane.add(jl);
    }
    public void itemStateChanged(ItemEvent ie)
    {
        String s = (String)ie.getItem();
        jl.setIcon(new ImageIcon(s + ".jpg"));
    }
}
```



Tabbed Panes

A tabbed pane is a component that appears as a group of folders in a file cabinet. Each folder has a title. When a user selects a folder, its contents become visible. Only one of the folders may be selected at a time. Tabbed panes are commonly used for setting configuration options.

Tabbed panes are encapsulated by the JTabbedPane class, which extends JComponent. There are three constructors of JTabbedPane.

```
JTabbedPane()
JTabbedPane(int tabPlacement)
JTabbedPane(int tabPlacement, int tabLayoutPolicy)
```

The first form creates an empty TabbedPane with a default tab placement of JTabbedPane.TOP. Second form creates an empty TabbedPane with the specified tab placement of any of the following:

```
JTabbedPane.TOP
JTabbedPane.BOTTOM
JTabbedPane.LEFT
JTabbedPane.RIGHT
```

The third form of constructor creates an empty TabbedPane with the specified tab placement and tab layout policy. Tab placements are listed above. Tab layout policy may be either of the following:

```
JTabbedPane.WRAP_TAB_LAYOUT
JTabbedPane.SCROLL_TAB_LAYOUT
```

Wrap tab policy

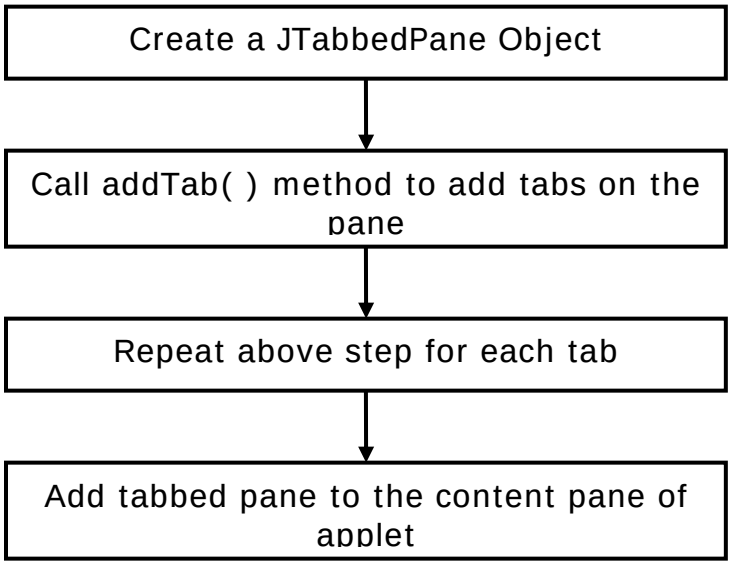
Scroll Tab Policy

Tabs are defined via the following method:

```
void addTab(String str, Component comp)
```

Here, *str* is the title for the tab, and comp is the component that should be added to the tab. Typically, a JPanel or a subclass of it is added. The general procedure to use a tabbed pane in an applet is outlined here:

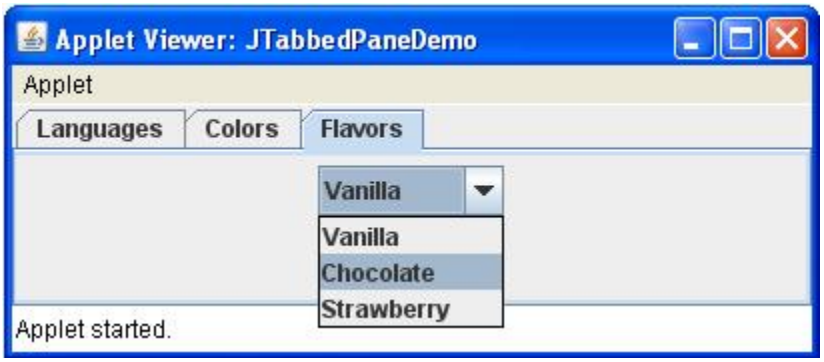
1. Create a JTabbedPane object.
2. Call addTab( ) to add a tab to the pane. (The arguments to this method define the title of the tab and the component it contains.)
3. Repeat step 2 for each tab.
4. Add the tabbed pane to the content pane of the applet.



The following example illustrates how to create a tabbed pane. The first tab is titled Languages and contains four buttons. Each button displays the name of a language. The second tab is titled Colors and contains three check boxes. Each check box displays the name of a color. The third tab is titled Flavors and contains one combo box. This enables the user to select one of three flavors.

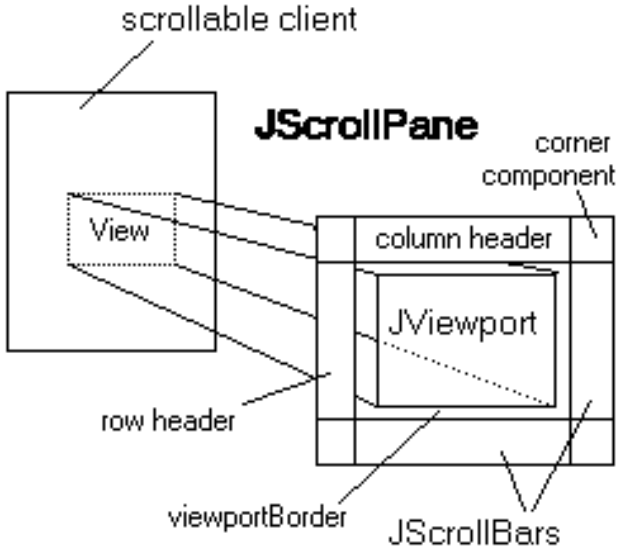
```
import javax.swing.*;
/*
<applet code="JTabbedPaneDemo" width=400 height=100>
</applet>
*/
```

```
public class JTabbedPaneDemo extends JApplet
{
    public void init()
    {
        JTabbedPane jtp = new JTabbedPane();
        jtp.addTab("Languages", new LangPanel());
        jtp.addTab("Colors", new ColorsPanel());
        jtp.addTab("Flavors", new FlavorsPanel());
        getContentPane().add(jtp);
    }
}
class LangPanel extends JPanel
{
    public LangPanel()
    {
        JButton b1 = new JButton("Marathi");
        add(b1);
        JButton b2 = new JButton("Hindi");
        add(b2);
        JButton b3 = new JButton("Bengali");
        add(b3);
        JButton b4 = new JButton("Tamil");
        add(b4);
    }
}
class ColorsPanel extends JPanel
{
    public ColorsPanel()
    {
        JCheckBox cb1 = new JCheckBox("Red");
        add(cb1);
        JCheckBox cb2 = new JCheckBox("Green");
        add(cb2);
        JCheckBox cb3 = new JCheckBox("Blue");
        add(cb3);
    }
}
class FlavorsPanel extends JPanel
{
    public FlavorsPanel()
    {
        JComboBox jcb = new JComboBox();
        jcb.addItem("Vanilla");
        jcb.addItem("Chocolate");
        jcb.addItem("Strawberry");
        add(jcb);
    }
}
```



Scroll Panes

A scroll pane is a component that presents a rectangular area in which a component may be viewed. Horizontal and/or vertical scroll bars may be provided if necessary. Scroll panes are implemented in Swing by the JScrollPane class, which extends JComponent.



Some of its constructors are shown here:

```
JScrollPane()
JScrollPane(Component comp)
JScrollPane(int vsb, int hsb)
JScrollPane(Component comp, int vsb, int hsb)
```

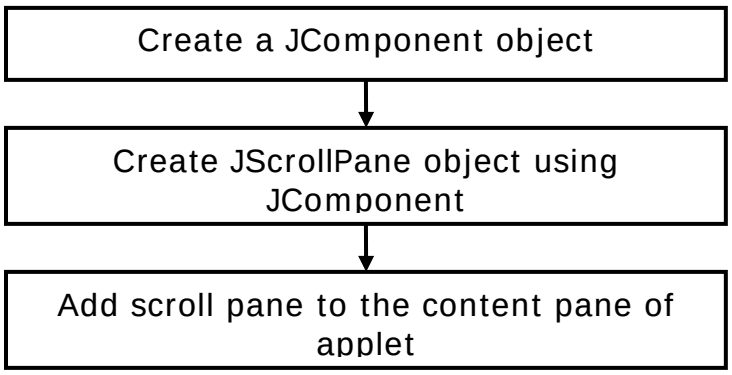
Here, *comp* is the component to be added to the scroll pane. *vsb* and *hsb* are int constants that define when vertical and horizontal scroll bars for this scroll pane are shown. These constants are defined by the **ScrollPaneConstants** interface. Some examples of these constants are described as follows:

Constant	**Description**
HORIZONTAL_SCROLLBAR_ALWAYS	Always provide horizontal scroll bar
HORIZONTAL_SCROLLBAR_AS_NEEDED	Provide horizontal scroll bar, if needed
VERTICAL_SCROLLBAR_ALWAYS	Always provide vertical scroll bar
VERTICAL_SCROLLBAR_AS_NEEDED	Provide vertical scroll bar, if needed

Here are the steps that you should follow to use a scroll pane in an applet:

1. Create a JComponent object.
2. Create a JScrollPane object. (The arguments to the constructor specify the component and the policies for vertical and horizontal scroll bars.)

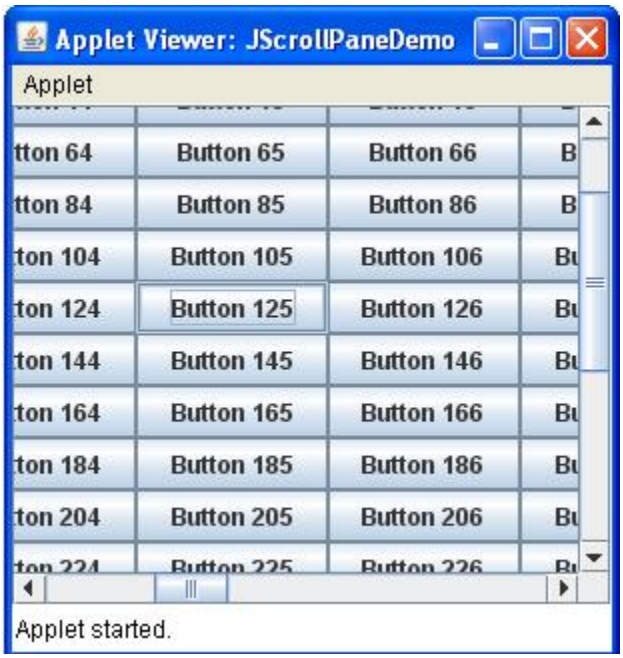
3. Add the scroll pane to the content pane of the applet.



The following example illustrates a scroll pane. First, the content pane of the JApplet object is obtained and a border layout is assigned as its layout manager. Next, a JPanel object is created and four hundred buttons are added to it, arranged into twenty columns. The panel is then added to a scroll pane, and the scroll pane is added to the content pane. This causes vertical and horizontal scroll bars to appear. We can use the scroll bars to scroll the buttons into view.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JScrollPaneDemo" width=300 height=250>
</applet>
*/
public class JScrollPaneDemo extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        JPanel jp = new JPanel();
        jp.setLayout(new GridLayout(20, 20));
        int b = 0;
        for(int i = 0; i < 20; i++)
        {
            for(int j = 0; j < 20; j++)
            {
                jp.add(new JButton("Button " + b));
                ++b;
            }
        }
        int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
        int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
        JScrollPane jsp = new JScrollPane(jp, v, h);
        contentPane.add(jsp, BorderLayout.CENTER);
    }
```

```
}
```



Trees

A tree is a component that presents a hierarchical view of data. A user has the ability to expand or collapse individual sub-trees in this display.

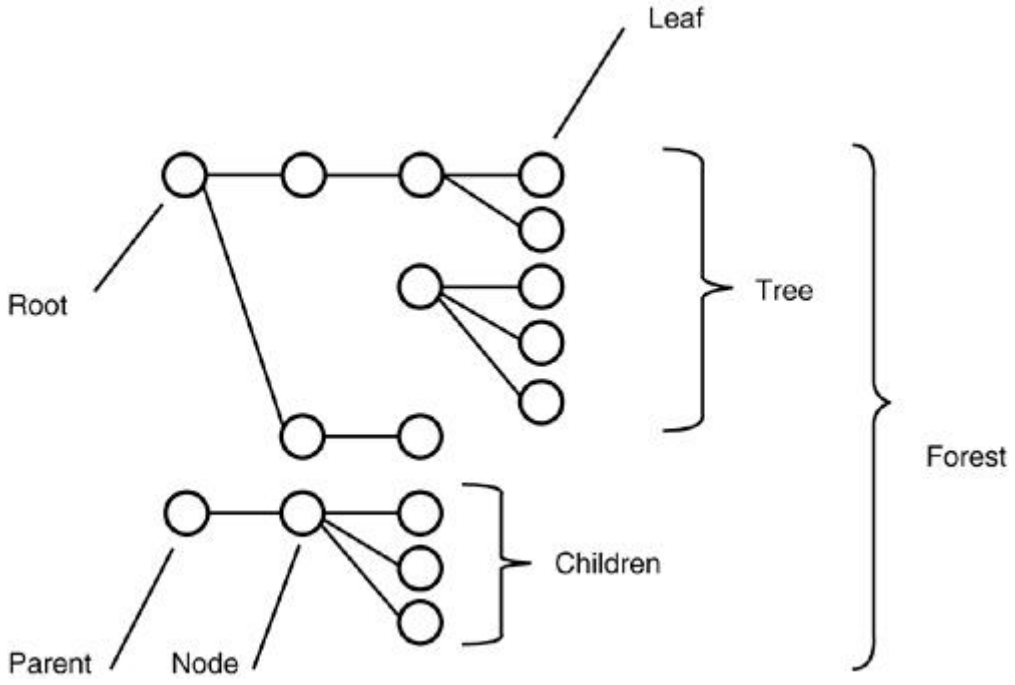


Fig. Tree Terminology

Trees are implemented in Swing by the JTree class, which extends JComponent. Some of its constructors are shown here:

```
JTree(Hashtable ht)
JTree(Object obj[ ])
JTree(TreeNode tn)
JTree(Vector v)
```

The first form creates a tree in which each element of the hash table *ht* is a child node. Each element of the array *obj* is a child node in the second form. The tree node *tn* is the root of the tree in the third form. Finally, the last form uses the elements of vector *v* as child nodes. A JTree object generates events when a node is expanded or collapsed. The addTreeExpansionListener( ) and removeTreeExpansionListener( ) methods allow listeners to register and unregister for these notifications. The signatures of these methods are shown here:

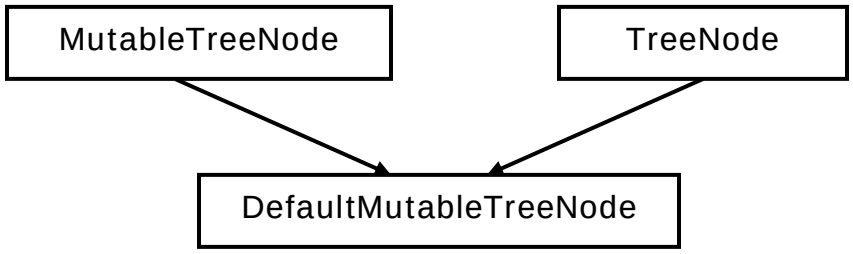
```
void addTreeExpansionListener(TreeExpansionListener tel)
void removeTreeExpansionListener(TreeExpansionListener tel)
```

Here, *tel* is the listener object. The getPathForLocation( ) method is used to translate a mouse click on a specific point of the tree to a tree path. Its signature is shown here:

```
TreePath getPathForLocation(int x, int y)
```

Here, *x* and *y* are the coordinates at which the mouse is clicked. The return value is a TreePath object that encapsulates information about the tree node that was selected by the user. The TreePath class encapsulates information about a path to a particular node in a tree. It provides several constructors and methods.

The TreeNode interface declares methods that obtain information about a tree node. For example, it is possible to obtain a reference to the parent node or an enumeration of the child nodes. The MutableTreeNode interface extends TreeNode. It declares methods that can insert and remove child nodes or change the parent node.



The DefaultMutableTreeNode class implements the MutableTreeNode interface. It represents a node in a tree. One of its constructors is shown here:

```
DefaultMutableTreeNode(Object obj)
```

Here, *obj* is the object to be enclosed in this tree node. The new tree node doesn't have a parent or children. To create a hierarchy of tree nodes, the add( ) method of DefaultMutableTreeNode can be used. Its signature is shown here:

```
void add(MutableTreeNode child)
```

Here, child is a mutable tree node that is to be added as a child to the current node.

Tree expansion events are described by the class TreeExpansionEvent in the javax.swing.event package. The getPath( ) method of this class returns a TreePath object that describes the path to the changed node. Its signature is shown here:

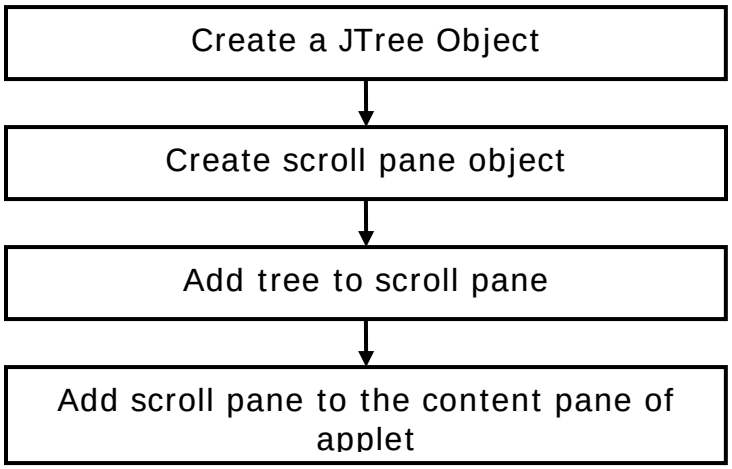
```
TreePath getPath( )
```

The TreeExpansionListener interface provides the following two methods:

```
void treeCollapsed(TreeExpansionEvent tee)
void treeExpanded(TreeExpansionEvent tee)
```

Here, *tee* is the tree expansion event. The first method is called when a sub-tree is hidden, and the second method is called when a sub-tree becomes visible. Here are the steps that we should follow to use a tree in an applet:

1. Create a JTree object.
2. Create a JScrollPane object. (The arguments to the constructor specify the tree and the policies for vertical and horizontal scroll bars.)
3. Add the tree to the scroll pane.
4. Add the scroll pane to the content pane of the applet.

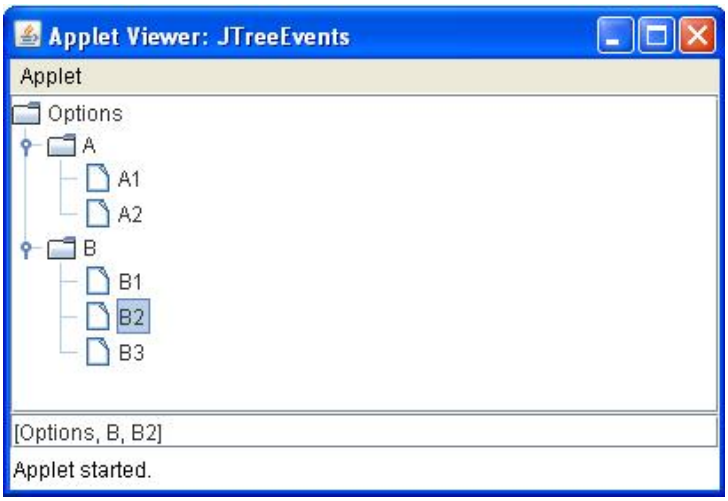


The following example illustrates how to create a tree and recognize mouse clicks on it. The init( ) method gets the content pane for the applet. A DefaultMutableTreeNode object labeled Options is created. This is the top node of the tree hierarchy. Additional tree nodes are then created, and the add( ) method is called to connect these nodes to the tree. A reference to the top node in the tree is provided as the argument to the JTree constructor. The tree is then provided as the argument to the JScrollPane constructor. This scroll pane is then added to the applet. Next, a text field is created and added to the applet. Information about mouse click events is presented in this text field. To receive mouse events from the tree, the addMouseListener( ) method of the JTree object is called. The argument to this method is an anonymous inner class that extends MouseAdapter and overrides the mouseClicked( ) method.

The doMouseClicked( ) method processes mouse clicks. It calls getPathForLocation( ) to translate the coordinates of the mouse click into a TreePath object. If the mouse is clicked at a point that does not cause a node selection, the return value from this method is null. Otherwise, the tree path can be converted to a string and presented in the text field.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;
/*
<applet code="JTreeEvents" width=400 height=200>
</applet>
*/
public class JTreeEvents extends JApplet
{
    JTree tree;
    JTextField jtf;
    public void init()
    {
        Container contentPane=getContentPane();
        contentPane.setLayout(new BorderLayout());
        DefaultMutableTreeNode top=new
               DefaultMutableTreeNode("Options");
        DefaultMutableTreeNode a= new DefaultMutableTreeNode("A");
        top.add(a);
        DefaultMutableTreeNode a1=new DefaultMutableTreeNode("A1");
        a.add(a1);
        DefaultMutableTreeNode a2=new DefaultMutableTreeNode("A2");
        a.add(a2);
        DefaultMutableTreeNode b= new DefaultMutableTreeNode("B");
        top.add(b);
        DefaultMutableTreeNode b1=new DefaultMutableTreeNode("B1");
        b.add(b1);
        DefaultMutableTreeNode b2=new DefaultMutableTreeNode("B2");
```

```
        b.add(b2);
        DefaultMutableTreeNode b3=new DefaultMutableTreeNode("B3");
        b.add(b3);
        tree=new JTree(top);
        int v=ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
        int h=ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
        JScrollPane jsp=new JScrollPane(tree,v,h);
        contentPane.add(jsp,BorderLayout.CENTER);
        jtf=new JTextField("",20);
        contentPane.add(jtf,BorderLayout.SOUTH);
        tree.addMouseListener(new MouseAdapter()
        {
            public void mouseClicked(MouseEvent me)
            {
                doMouseClicked(me);
            }
        });
    }
    void doMouseClicked(MouseEvent me)
    {
        TreePath tp=tree.getPathForLocation(me.getX(),me.getY());
        if(tp!=null)
            jtf.setText(tp.toString());
        else
            jtf.setText("");
    }
}
```



The string presented in the text field describes the path from the top tree node to the selected node.

Tables

A table is a component that displays rows and columns of data. We can drag the cursor on column boundaries to resize columns. We can also drag a column to a new position. Tables are implemented by the JTable class, which extends JComponent. One of its constructors is shown here:

```
JTable(Object data[ ][ ], Object colHeads[ ])
JTable(int numRows, int numColumns)
JTable(Vector rowData, Vector columnData)
```

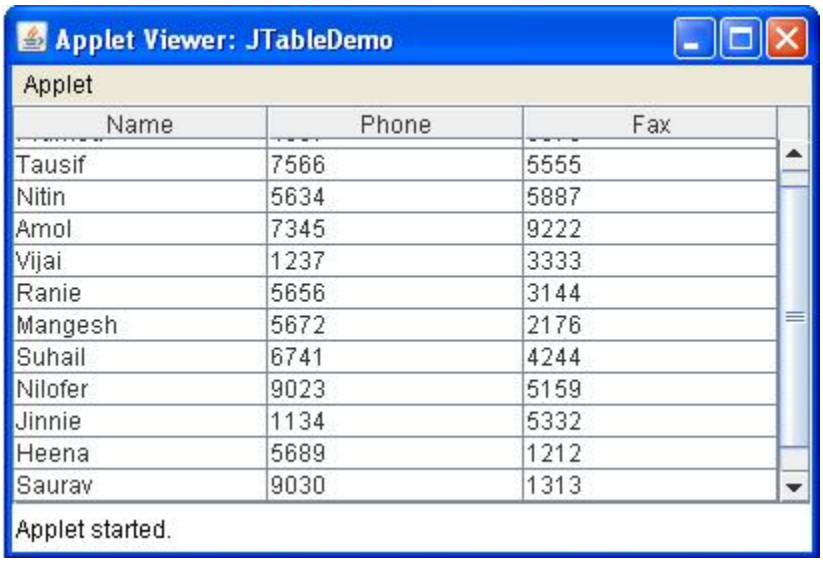
Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings. The '*numRows*' and '*numColumns*' are values with which the table is to be created. The '*rowData*' and '*columnData*' are the vector values by which the table is constructed.
Here are the steps for using a table in an applet:

1) Create a JTable object.
2) Create a JScrollPane object. (The arguments to the constructor specify the table and the policies for vertical and horizontal scroll bars.)
3) Add the table to the scroll pane.
4) Add the scroll pane to the content pane of the applet.

The following example illustrates how to create and use a table. The content pane of the JApplet object is obtained and a border layout is assigned as its layout manager. A one-dimensional array of strings is created for the column headings. This table has three columns. A two-dimensional array of strings is created for the table cells. We can see that each element in the array is an array of three strings. These arrays are passed to the JTable constructor. The table is added to a scroll pane and then the scroll pane is added to the content pane.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTableDemo" width=400 height=200>
</applet>
*/
public class JTableDemo extends JApplet
{
    public void init()
    {
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        final String[] colHeads = { "Name", "Phone", "Fax" };
        final Object[][] data = {
```

```
            { "Pramod", "4567", "8675" },
            { "Tausif", "7566", "5555" },
            { "Nitin", "5634", "5887" },
            { "Amol", "7345", "9222" },
            { "Vijai", "1237", "3333" },
            { "Ranie", "5656", "3144" },
            { "Mangesh", "5672", "2176" },
            { "Suhail", "6741", "4244" },
            { "Nilofer", "9023", "5159" },
            { "Jinnie", "1134", "5332" },
            { "Heena", "5689", "1212" },
            { "Saurav", "9030", "1313" },
            { "Raman", "6751", "1415" }
    };
    JTable table = new JTable(data, colHeads);
    int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
    int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
    JScrollPane jsp = new JScrollPane(table, v, h);
    contentPane.add(jsp, BorderLayout.CENTER);
    }
}
```



References

1. **Java 2 the Complete Reference**,
   Fifth Edition by Herbert Schildt, 2001 Osborne McGraw Hill
   Chapter 26: The Tour of Swing
   (Most of the data is referred from this book)

2. **Java 6 Black Book,**
   Kogent Solutions, 2007, Dreamtech Press
   Chapter 15: Swing–Applets, Applications and Pluggable Look and Feel.

3. **JDK 5.0 Documentation,**
   Sun Microsystems, USA, www.java.sun.com

Chapter 05
Servlets

Lectures allotted: 08
Marks Given: 16

Contents:

5.1 Background,
The Life Cycle of a Servlet,
The Java Servlet Development Kit,
The Simple Servlet,
The Servlet API
The javax.Servlet Package,
Reading Servlet Parameters,
Reading Initialization Parameters
5.2 The javax.servlet.http package,
Handling HTTP Requests and responses
5.3 Using Cookies,
Session Tracking,
Security Issues,
Exploring Servlet

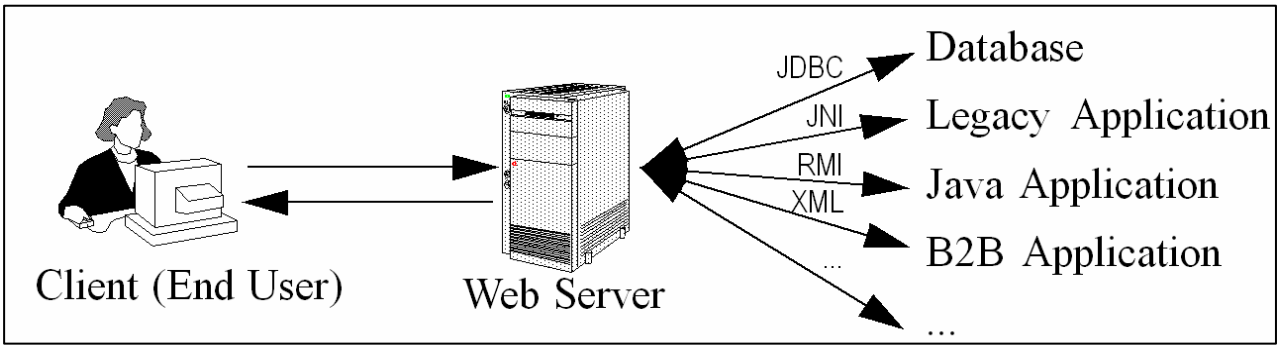
Introduction[Ref.1]

Servlets are small programs that execute on the server side of a Web connection. Just as applets dynamically extend the functionality of a Web browser, servlets dynamically extend the functionality of a Web server.

Servlets are generic extensions to Java-enabled servers. They are secure, portable, and easy to use replacement for CGI. Servlet is a dynamically loaded module that services requests from a Web server and executed within the Java Virtual Machine. Because the servlet is running on the server side, it does not depend on browser compatibility.

Servlet's Job

- Read explicit data sent by client (form data)
- Read implicit data sent by client (request headers)
- Generate the results
- Send the explicit data back to client (HTML)
- Send the implicit data to client (status codes and response headers)



The Hypertext Transfer Protocol (HTTP) [Ref.1]

HTTP is the protocol that allows web servers and browsers to exchange data over the web. It is a request and response protocol. The client requests a file and the server responds to the request. HTTP uses reliable TCP connections—by default on TCP port 80. HTTP (currently at version 1.1 at the time of this writing) was first defined in RFC 2068. It was then refined in RFC 2616, which can be found at http://www.w3c.org/Protocols/.

In HTTP, it's always the client who initiates a transaction by establishing a connection and sending an HTTP request. The server is in no position to contact a client or make a callback connection to the client. Either the client or the server can prematurely terminate a connection. For example, when using a web browser we can click the Stop button on our browser to stop the download process of a file, effectively closing the HTTP connection with the web server.

HTTP Requests [Ref.1]

An HTTP transaction begins with a request from the client browser and ends with a response from the server. An HTTP request consists of three components:

- Method——URI—Protocol/Version
- Request headers
- Entity body

An example of an HTTP request is the following:

```
GET /servlet/default.jsp HTTP/1.1
Accept: text/plain; text/html
Accept-Language: en-gb
Connection: Keep-Alive
Host: localhost
Referer: http://localhost/ch8/SendDetails.htm
User-Agent: Mozilla/4.0 (compatible; MSIE 4.01; Windows 98)
Content-Length: 33
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate

LastName=Franks&FirstName=Michael
```

The method—URI—protocol version appears as the first line of the request.

GET /servlet/default.jsp HTTP/1.1
where GET is the request method, /servlet/default.jsp represents the URI and HTTP/1.1 the Protocol/Version section.

The request method will be explained in more details in the next section, "HTTP request Methods."

The URI specifies an Internet resource completely. A URI is usually interpreted as being relative to the server's root directory. Thus, it should always begin with a forward slash /. A URL is actually a type of URI (see http://www.ietf.org/rfc/rfc2396.txt). The Protocol version represents the version of the HTTP protocol being used.

The request header contains useful information about the client environment and the entity body of the request. For example, it could contain the language the browser is set for, the length of the entity body, and so on. Each header is separated by a carriage return/linefeed (CRLF) sequence.

Between the headers and the entity body, there is a blank line (CRLF) that is important to the HTTP request format. The CRLF tells the HTTP server where the entity body begins. In some Internet programming books, this CRLF is considered the fourth component of an HTTP request.

In the previous HTTP request, the entity body is simply the following line:

LastName=Franks&FirstName=Michael
The entity body could easily become much longer in a typical HTTP request.

Method	Description
GET	GET is the simplest, and probably, most used HTTP method. GET simply retrieves the data identified by the URL. If the URL refers to a script (CGI, servlet, and so on), it returns the data produced by the script.
HEAD	The HEAD method provides the same functionality as GET, but HEAD only returns HTTP headers without the document body.
POST	Like GET, POST is also widely used. Typically, POST is used in HTML forms. POST is used to transfer a block of data to the server in the entity body of the request.
OPTIONS	The OPTIONS method is used to query a server about the capabilities it provides. Queries can be general or specific to a particular resource.
PUT	The PUT method is a complement of a GET request, and PUT stores the entity body at the location specified by the URI. It is similar to the PUT function in FTP.
DELETE	The DELETE method is used to delete a document from the server. The document to be deleted is indicated in the URI section of the request.
TRACE	The TRACE method is used to tract the path of a request through firewall and multiple proxy servers. TRACE is useful for debugging complex network problems and is similar to the traceroute tool.

Of the seven methods, only GET and POST are commonly used in an Internet application.

HTTP Responses [Ref.1]

Similar to requests, an HTTP response also consists of three parts:

- Protocol—Status code—Description
- Response headers
- Entity body

The following is an example of an HTTP response:

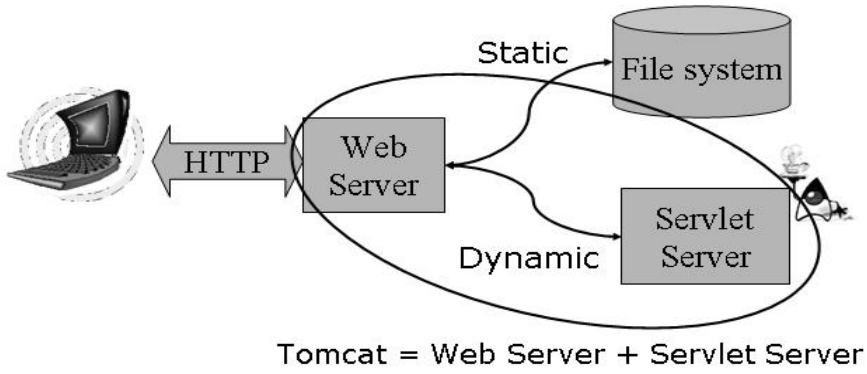
```
HTTP/1.1 200 OK
Server: Microsoft-IIS/4.0
Date: Mon, 3 Jan 1998 13:13:33 GMT
Content-Type: text/html
Last-Modified: Mon, 11 Jan 1998 13:23:42 GMT
Content-Length: 112
```

```
<HTML>
<HEAD>
<TITLE>HTTP Response Example</TITLE></HEAD><BODY>
Welcome to Brainy Software
</BODY>
</HTML>
```

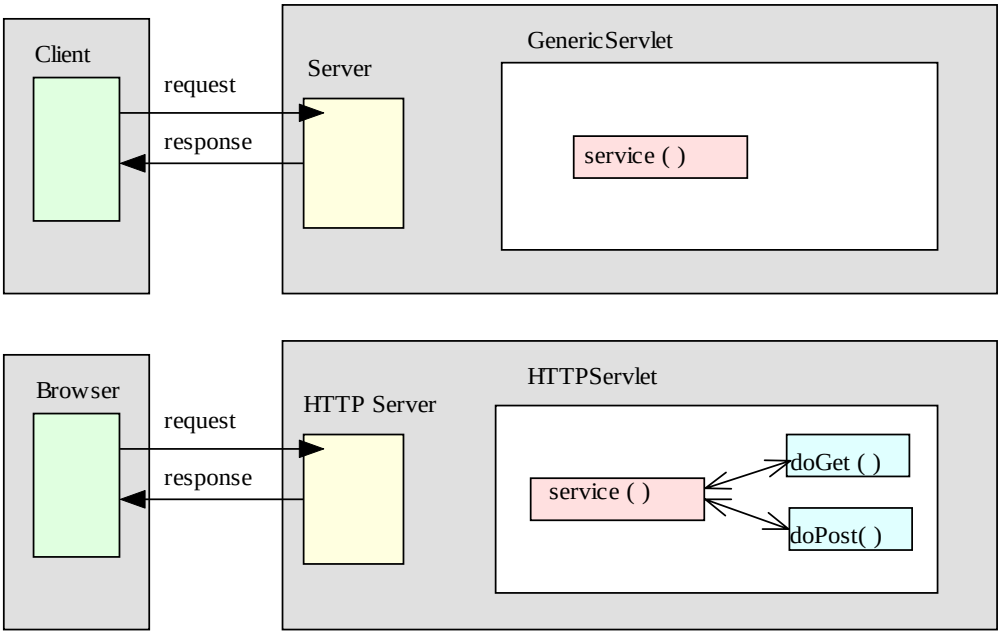
The first line of the response header is similar to the first line of the request header. The first line tells you that the protocol used is HTTP version 1.1, the request succeeded (200 = success), and that everything went okay.

The response headers contain useful information similar to the headers in the request. The entity body of the response is the HTML content of the response itself. The headers and the entity body are separated by a sequence of CRLFs.

Where are servlets?



Servlet Application Architecture



Applications of Java Servlets

- Building e-commerce store fronts
  - Servlet builds an online catalog based on the contents of a database
  - Customer places an order, which is processed by another servlet
- Servlets as wrappers for legacy systems
- Servlets interacting with EJB applications

Java Servlet alternatives [Ref.1]

ColdFusion. Allaire's ColdFusion provides HTML-like custom tags that can be used to perform a number of operations, especially querying a database. This technology had its glamorous time in the history of the World Wide Web as the main technology for web application programming. Its glorious time has since gone with the invention of other technologies.

Server-side JavaScript (SSJS). SSJS is an extension of the JavaScript language, the scripting language that still rules client-side web programming. SSJS can access Java classes deployed at the server side using the LiveWire technology from Netscape.

PHP. PHP is an exciting open-source technology that has matured in recent years. The technology provides easy web application development with its session management and includes some built-in functionality, such as file upload. The number of programmers embracing PHP as their technology of choice has risen sharply in recent years.

Servlet. The servlet technology was introduced by Sun Microsystems in 1996.

JavaServer Pages (JSP). JSP is an extension of the servlet technology.

Active Server Pages (ASP). Microsoft's ASP employs scripting technologies that work in Windows platforms, even though there have been efforts to port this technology to other operating systems. Windows ASP works with the Internet Information Server web server. This technology will soon be replaced by Active Server Pages.NET.

Active Server Pages.NET (ASP.NET). This technology is part of Microsoft's .NET initiative. Interestingly, the .NET Framework employs a runtime called the Common Language Runtime that is very similar to Java Virtual Machine and provides a vast class library available to all .NET languages and from ASP.NET pages. ASP.NET is an exciting technology. It introduced several new technologies including state management that does not depend on cookies or URL rewriting.

The Benefits of Servlets

- *Efficiency*: More efficient – uses lightweight java threads as opposed to individual processes.
- *Persistency*: Servlets remain in memory. They can maintain state between requests.
- *Portability*: Since servlets are written in Java, they are platform independent.
- *Robustness*: Error handling, Garbage collector to prevent problems with memory leaks. Large class library – network, file, database, distributed object components, security, etc.
- *Extensibility*: Creating new subclasses that suite your needs Inheritance, polymorphism, etc.
- *Security*: Security provided by the server as well as the Java Security Manager. It eliminates problems associated with executing cgi scripts using operating system "shells".
- *Powerful*: Servlets can directly talk to web server and facilitates database connection pooling, session tracking etc.
- *Convenient*: Parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, etc.
- *Rapid development cycle:* As a Java technology, servlets have access to the rich Java library, which helps speed up the development process.
- *Widespread acceptance:* Java is a widely accepted technology. This means that numerous vendors work on Java-based technologies. One of the advantages of this widespread acceptance is that we can easily find and purchase components that suit our needs, which saves precious development time.

How a Servlet Works [Ref.1]

A servlet is loaded by the servlet container the first time the servlet is requested. The servlet then is forwarded the user request, processes it, and returns the response to the servlet container, which in turn sends the response back to the user. After that, the servlet stays in memory waiting for other requests—it will not be unloaded from the memory unless the servlet container sees a shortage of memory. Each time the servlet is requested, however, the servlet container compares the timestamp of the loaded servlet with the servlet class file. If the class file timestamp is more recent, the servlet is reloaded into memory. This way, we don't need to restart the servlet container every time we update our servlet.

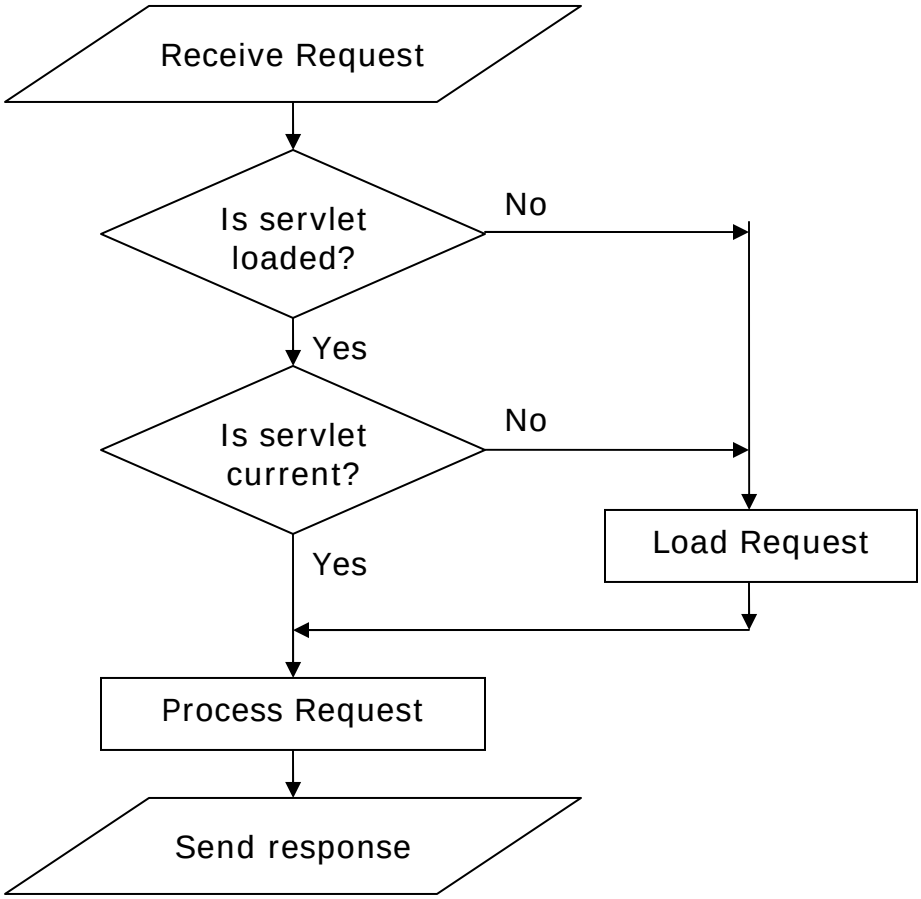


Fig. How servlet works?

The Tomcat Servlet Container [Ref.1]

A number of servlet containers are available today these are listed below:

- Apache Tomcat
  http://jakarta.apache.org/tomcat/
- Allaire/Macromedia JRun
  http://www.macromedia.com/software/jrun/
- New Atlanta ServletExec
  http://www.servletexec.com/
- Gefion Software LiteWebServer
  http://www.gefionsoftware.com/LiteWebServer/
- Caucho's Resin
  http://www.caucho.com/

The most popular one—and the one recognized as the official servlet/JSP container—is Apache Tomcat. Originally designed by Sun Microsystems, Tomcat source code was handed over to the Apache Software Foundation in October 1999. In this new home, Tomcat was included as part of the Jakarta Project,

one of the projects of the Apache Software Foundation. Working through the Apache process, Apache, Sun, and other companies—with the help of volunteer programmers worldwide—turned Tomcat into a world-class servlet reference implementation. Currently we are using Apache Tomcat version 6.0.18.

Tomcat by itself is a web server. This means that you can use Tomcat to service HTTP requests for servlets, as well as static files (HTML, image files, and so on). In practice, however, since it is faster for non-servlet, non-JSP requests, Tomcat normally is used as a module with another more robust web server, such as Apache web server or Microsoft Internet Information Server. Only requests for servlets or JSP pages are passed to Tomcat.

For writing a servlet, we need at Java Development Kit installed on our computer. Tomcat is written purely in Java.

Steps to Running Your First Servlet [Ref.1]

After we have installed and configured Tomcat, we can put it into service. Basically, we need to follow steps to go from writing our servlet to running it. These steps are summarized as follows:

- Write the servlet source code. We need to import the javax.servlet package and the javax.servlet.http package in your source file.
- Compile your source code.
- Create a deployment descriptor.
- Run Tomcat.
- Call your servlet from a web browser.

1. Write the Servlet Source Code

In this step, we prepare our source code. We can write the source code ourself using any text editor.

The following program shows a simple servlet called TestingServlet. The file is named TestingServlet.java. The servlet sends a few HTML tags and some text to the browser.

```
//TestingServlet.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class TestingServlet extends HttpServlet
{
  public void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException
    {
```

```
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>Servlet Testing</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("Welcome to the Servlet Testing Center");
    out.println("</BODY>");
    out.println("</HTML>");
  }
}
```

Now, save TestingServlet.java file to /bin directory of JDK.

2. Compiling the source code

For our servlet source code to compile, we need to include the path to the servlet-api.jar file in our CLASSPATH environment variable. The servlet-api.jar is located in the C:\Program Files\Apache Software Foundation\Tomcat 6.0\ directory. Here, the drive name depends upon our selection while installation of Tomcat 6.0 on computer. So, compile the file using following way:

```
javac TestingServlet.java -classpath "G:\Program Files\Apache
Software Foundation\Tomcat 6.0\lib\servlet-api.jar"
```

After successful compilation, we will get a class file named TestingServlet.class. Now, copy that class file into directory \classes under web-inf as shown in the following figure. All the servlet classes resides in this directory.

3. Create the Deployment Descriptor

A deployment descriptor is an optional component in a servlet application. The descriptor takes the form of an XML document called *web.xml* and must be located in the WEB-INF directory of the servlet application. When present, the deployment descriptor contains configuration settings specific to that application. In order to create the deployment descriptor, we now need to create or edit a web.xml file and place it under the WEB-INF directory. The *web.xml* for this example application must have the following content.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <servlet>
    <servlet-name>Testing</servlet-name>
```

```
    <servlet-class>TestingServlet</servlet-class>
    <servlet-mapping>
         <servlet-name>TestingServlet</servlet-name>
         <url-pattern>/servlets/servlet/TestingServlet
              </url-pattern>
    </servlet-mapping>
  </servlet>
</web-app>
```

The web.xml file has one element—web-app. We should write all our servlets under <web-app>. For each servlet, we have a <servlet> element and we need the <servlet-name> and <servlet-class> elements. The <servlet-name> is the name for our servlet, by which it is known Tomcat. The <servlet-class> is the compiled file of your servlet without the .class extension.

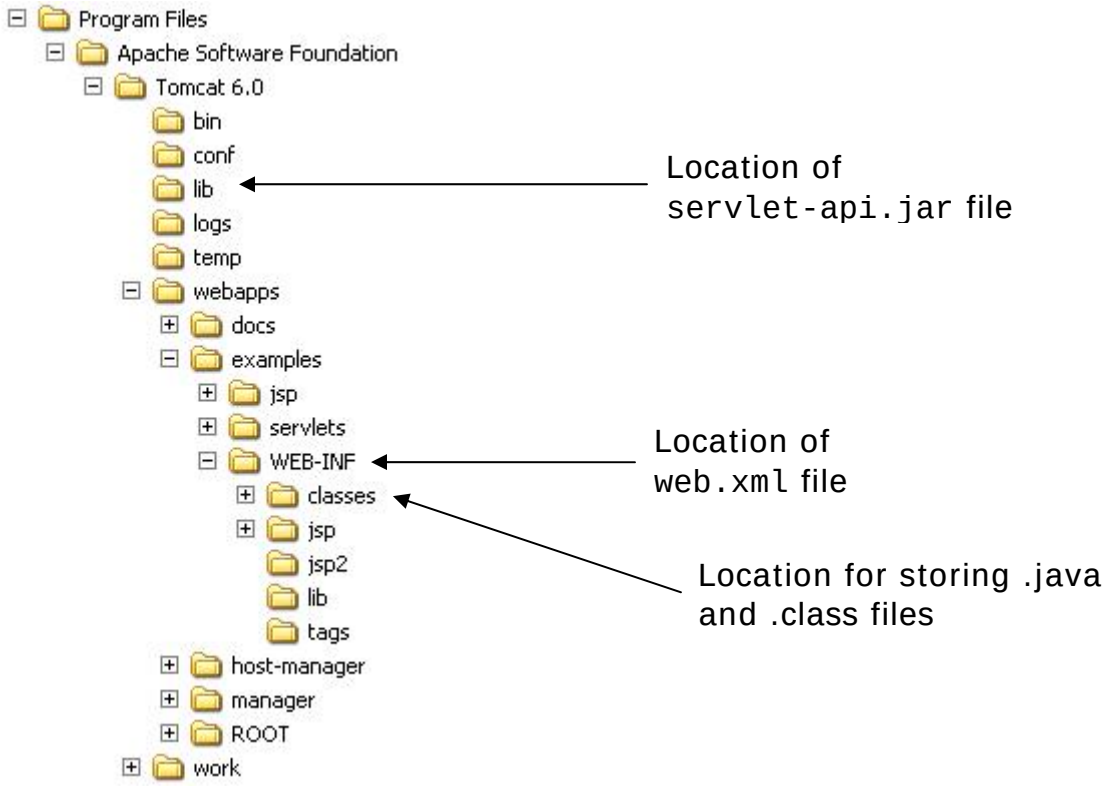


Fig. Apache Tomcat 6.0 Directory Structure.

We can also add multiple servlet names in our file using multiple servlet tags. The url-pattern suggests the url by which we are going to class our servlet in the web browser. Instead of doing this we can just modify the contents of web.xml by adding the names and mapping of our servlet code.

4. Run Tomcat

If Tomcat is not already running, we need to start it by selecting the option “monitor tomcat” from srart menu. We will find the icon of Apache Tomcat on the taskbar when it is running.

5. Call Your Servlet from a Web Browser

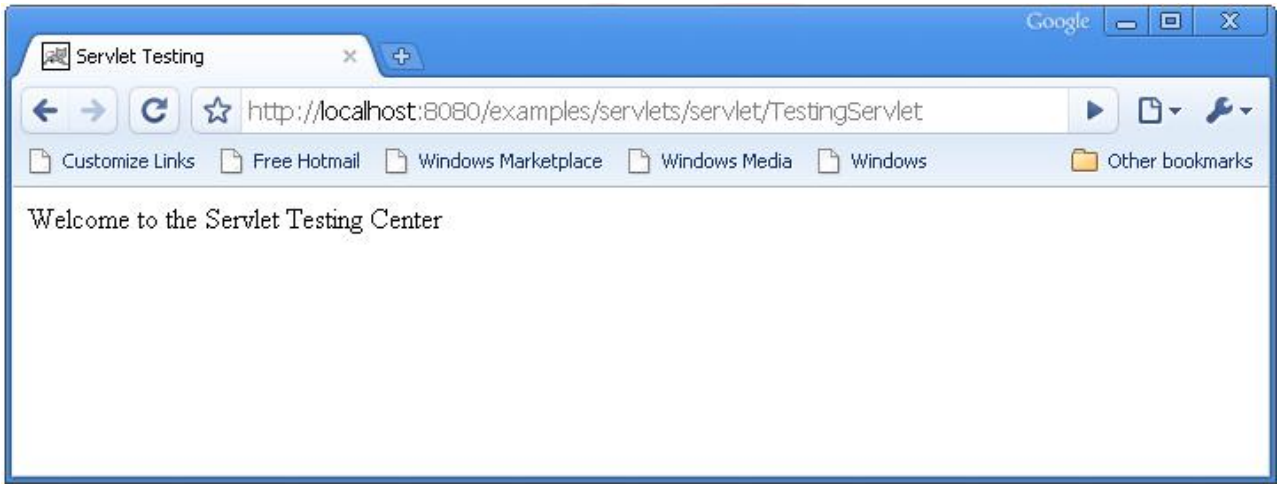
Now, we can call our servlet from a web browser. By default, Tomcat runs on port 8080. The URL for that servlet has the following format:

http://domain-name/virtual-directory/servlet/servlet-name

If we run the web browser from the same computer as Tomcat, you can replace the domain-name part with "localhost". In that case, the URL for your servlet is:

http://localhost:8080/examples/servlets/servlet/TestingServlet

Typing the URL in the Address or Location box of our web browser will give you the string "Welcome to the Servlet Testing Center," as shown in Figure 1.5.



The javax.servlet package [Ref.1]

The javax.servlet package contains seven interfaces, three classes, and two exceptions. The seven interfaces are as follows:

- `RequestDispatcher`
- `Servlet`
- `ServletConfig`
- `ServletContext`
- `ServletRequest`

- ServletResponse
- SingleThreadModel

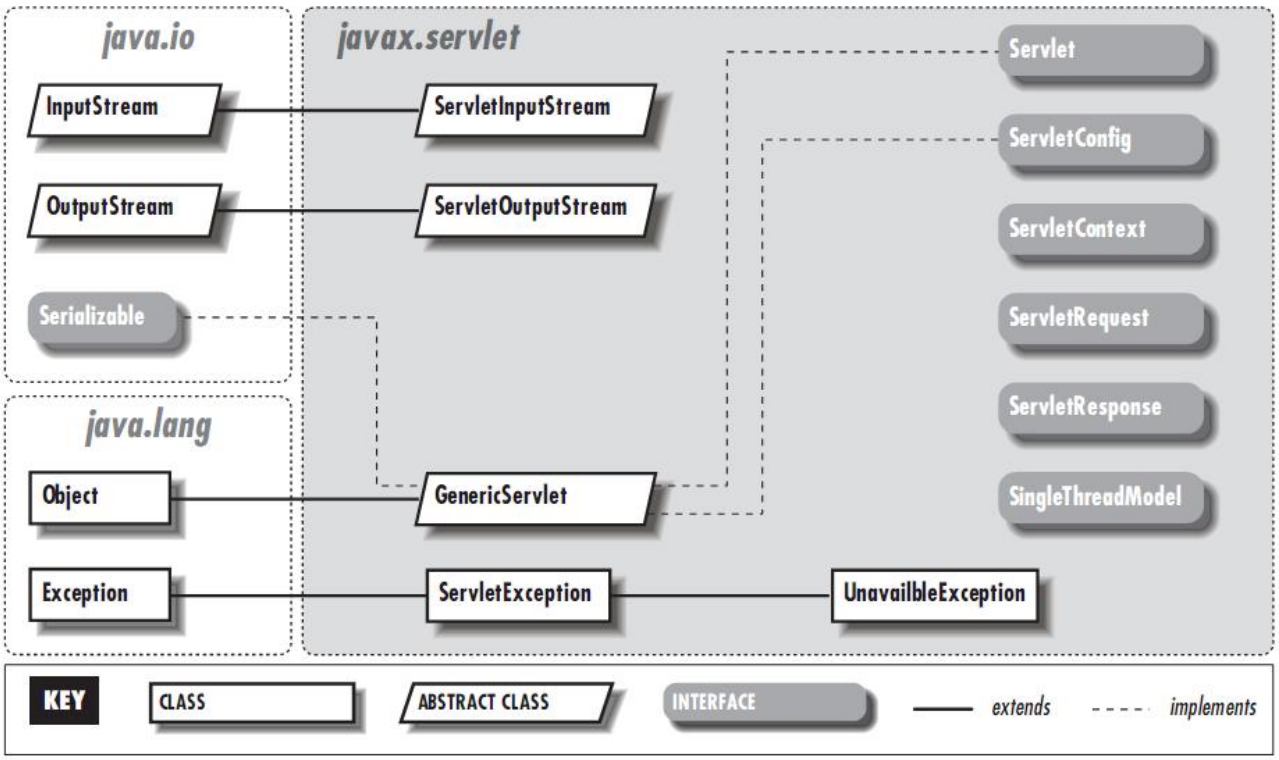
The three classes are as follows:

- GenericServlet
- ServletInputStream
- ServletOutputStream

And, finally, the exception classes are these:

- ServletException
- UnavailableException

The object model of the javax.servlet package is shown in figure below:



The javax.servlet package

The Servlet's Life Cycle

Applet life cycle contains methods: init( ), start( ), paint( ), stop( ), and destroy( ) – appropriate methods called based on user action. Similarly, servlets operate in the context of a request and response model managed by a servlet engine The engine does the following:

- Loads the servlet when it is first requested.
- Calls the servlet's init( ) method.
- Handles any number of requests by calling the servlet's service( ) method.
- When shutting down, calls each servlet's destroy( ) method.

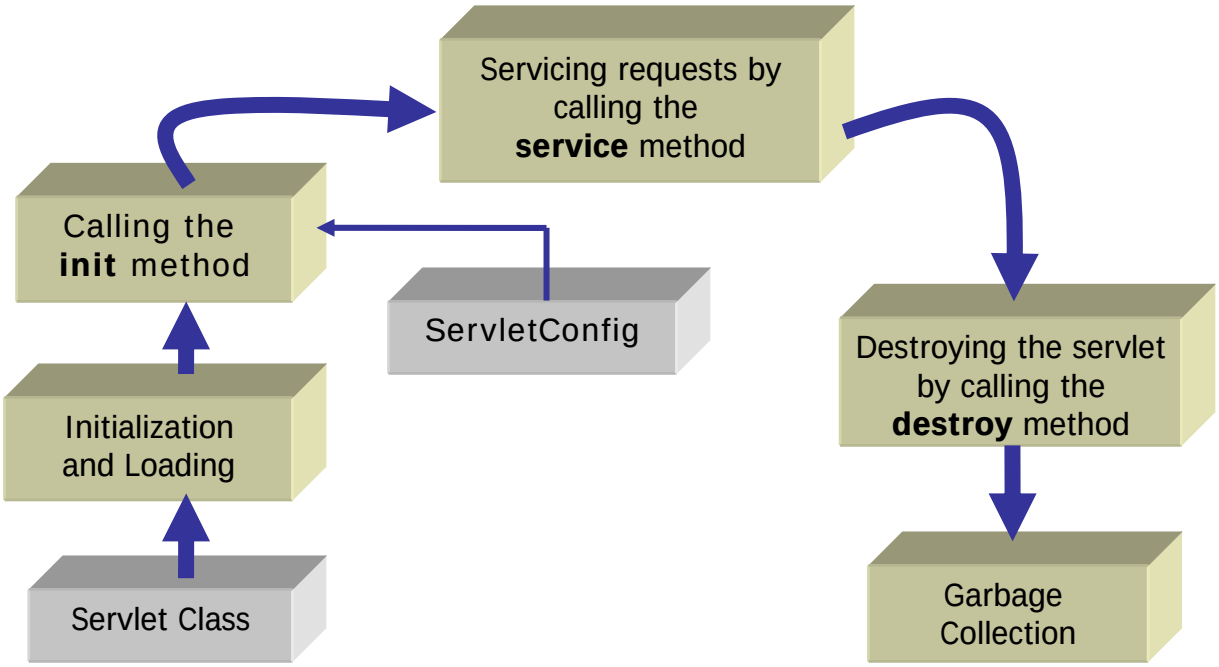


Fig. Servlet's Life Cycle

The init( ) method

- It request for a servlet received by the servlet engine.
- Checks to see if the servlet is already loaded.
- If not, uses a class loader to get the required servlet class and instantiates it by calling the constructor method.
- After the servlet is loaded, but before it services any requests, the init ( ) method is called.
- Inside init( ), the resources used by the servlet are initialized. E.g: establishing database connection.
- This method is called only once just before the servlet is placed into service.
- The init( ) method takes a ServletConfig object as a parameter. Its signature is:
  `public void init(ServletConfig config) throws ServletException`
- Most common way of doing this is to have it call the super.init( ) passing it the ServletConfig object.

The service( ) method

- The service( ) method handles all requests sent by a client.
- It cannot start servicing requests until the init( ) method has been executed.
- Only a single instance of the servlet is created and the servlet engine dispatches each request in a single thread.
- The service( ) method is used only when extending GenericServlet class.
- Since servlets are designed to operate in the HTTP environment, the HttpServlet class is extended.
- The service(HttpServletRequest, HttpServletResponse) method examines the request and calls the appropriate doGet() or doPost() method.
- A typical Http servlet includes overrides to one or more of these subsidiary methods rather than an override to service().

The destroy( ) method

- This method signifies the end of a servlet's life.
- The resources allocated during init( ) are released.
- Save persistent information that will be used the next time the servlet is loaded.
- The servlet engine unloads the servlet.
- Calling destroy( ) yourself will not acutally unload the servlet. Only the servlet engine can do this.

Demonstrating the Life Cycle of a Servlet [Ref.1]

The following program contains the code for a servlet named PrimitiveServlet, a very simple servlet that exists to demonstrate the life cycle of a servlet. The PrimitiveServlet class implements javax.servlet.Servlet (as all servlets must) and provides implementations for all the five methods of servlet. What it does is very simple. Each time any of the init, service, or destroy methods is called, the servlet writes the method's name to the console.

```
import javax.servlet.*;
import java.io.IOException;
public class PrimitiveServlet implements Servlet
{
  public void init(ServletConfig config) throws ServletException {
    System.out.println("init");
  }
  public void service(ServletRequest request,
                      ServletResponse response)
    throws ServletException, IOException {
    System.out.println("service");
  }
  public void destroy() {
```

```
    System.out.println("destroy");
  }
  public String getServletInfo() {
    return null;
  }
  public ServletConfig getServletConfig() {
    return null;
  }
}
```

The web.xml File for PrimitiveServlet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
  PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet>
    <servlet-name>PrimitiveServlet</servlet-name>
    <servlet-class>PrimitiveServlet</servlet-class>
  </servlet>
</web-app>
```

We should then be able to call this servlet from our browser by typing the following URL:

http://localhost:8080/examples/servlets/servlet/PrimitiveServlet

The first time the servlet is called, the console displays these two lines:

```
init
service
```

This tells us that the init method is called, followed by the service method. However, on subsequent requests, only the service method is called. The servlet adds the following line to the console:

```
service
```

This proves that the init method is called only once.

Requests and Responses [Ref.1]

Requests and responses are what a web application is all about. In a servlet application, a user using a web browser sends a request to the servlet container, and the servlet container passes the request to the servlet.

In a servlet paradigm, the user request is represented by the ServletRequest object passed by the servlet container as the first argument to the service method. The service method's second argument is a ServletResponse object, which represents the response to the user.

The ServletRequest Interface [Ref.1]

The ServletRequest interface defines an object used to encapsulate information about the user's request, including parameter name/value pairs, attributes, and an input stream.

The ServletRequest interface provides important methods that enable us to access information about the user. For example, the getParameterNames method returns an Enumeration containing the parameter names for the current request. In order to get the value of each parameter, the ServletRequest interface provides the getParameter method.

The getRemoteAddress and getRemoteHost methods are two methods that we can use to retrieve the user's computer identity. The first returns a string representing the IP address of the computer the client is using, and the second method returns a string representing the qualified host name of the computer.

The following example, shows a ServletRequest object in action. The example consists of an HTML form in a file named index.html and a servlet called RequestDemoServlet.

The index.html file:

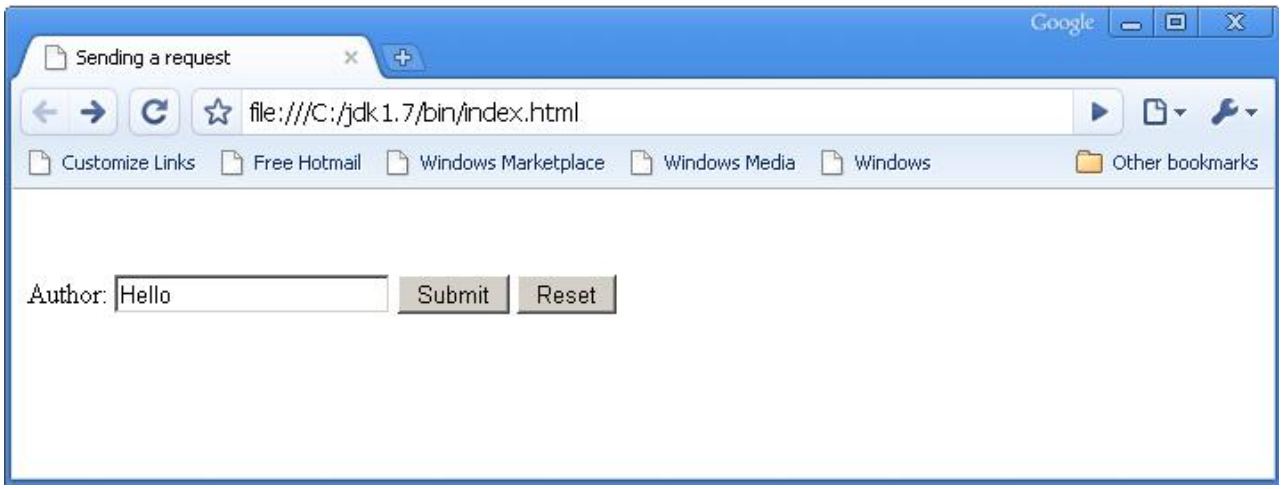
```
<HTML>
<HEAD>
<TITLE>Sending a request</TITLE>
</HEAD>
<BODY>
<FORM ACTION =
http://localhost:8080/examples/servlets/servlet/RequestDemoServlet
METHOD="POST">
<BR><BR>
Author: <INPUT TYPE="TEXT" NAME="Author">
<INPUT TYPE="SUBMIT" NAME="Submit">
<INPUT TYPE="RESET" VALUE="Reset">
</FORM>
</BODY>
</HTML>
```

The RequestDemoServlet.java file:

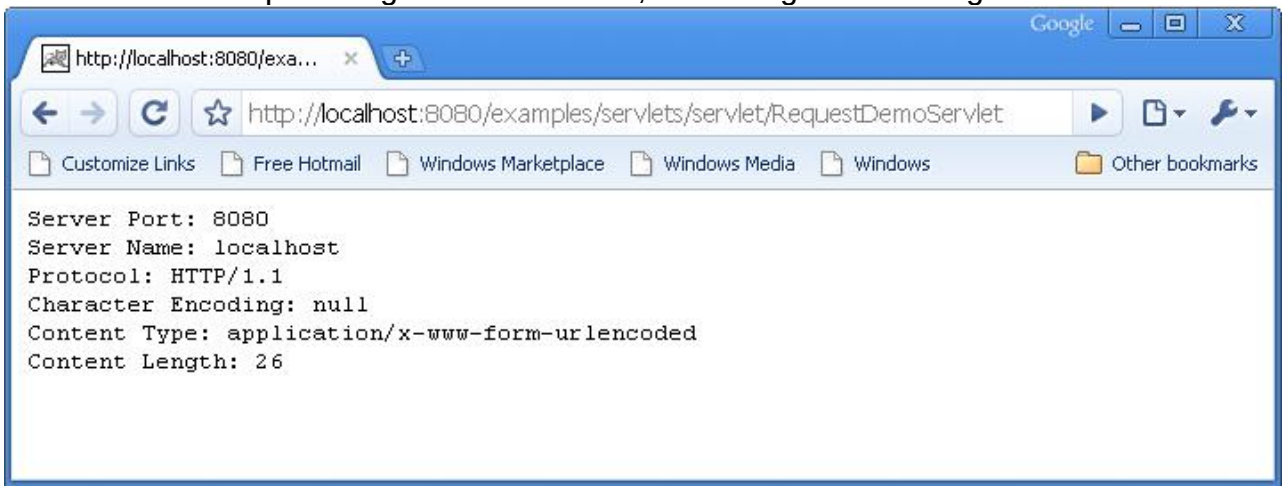
```
import javax.servlet.*;
import java.util.Enumeration;
import java.io.IOException;
public class RequestDemoServlet implements Servlet {
```

```
  public void init(ServletConfig config) throws ServletException {
  }
  public void destroy() {
  }
  public void service(ServletRequest request,
                      ServletResponse response)
    throws ServletException, IOException {
    System.out.println("Server Port: " + request.getServerPort());
    System.out.println("Server Name: " + request.getServerName());
    System.out.println("Protocol: " + request.getProtocol());
    System.out.println("Character Encoding: " +
      request.getCharacterEncoding());
    System.out.println("Content Type: " +
          request.getContentType());
    System.out.println("Content Length: " +
          request.getContentLength());
    System.out.println("Remote Address: " +
          request.getRemoteAddr());
    System.out.println("Remote Host: " + request.getRemoteHost());
    System.out.println("Scheme: " + request.getScheme());
    Enumeration parameters = request.getParameterNames();
    while (parameters.hasMoreElements()) {
      String parameterName = (String) parameters.nextElement();
      System.out.println("Parameter Name: " + parameterName);
      System.out.println("Parameter Value: " +
        request.getParameter(parameterName));
    }
    Enumeration attributes = request.getAttributeNames();
    while (attributes.hasMoreElements()) {
      String attribute = (String) attributes.nextElement();
      System.out.println("Attribute name: " + attribute);
      System.out.println("Attribute value: " +
        request.getAttribute(attribute));
    }
  }
  public String getServletInfo() {
    return null;
  }
  public ServletConfig getServletConfig() {
    return null;
  }
}
```

The snapshot of index.html:



After pressing submit button, we will get following window:



The ServletResponse Interface [Ref.1]

The ServletResponse interface represents the response to the user. The most important method of this interface is getWriter, from which we can obtain a java.io.PrintWriter object that we can use to write HTML tags and other text to the user.

The codes of the program given below offer an HTML file named index2.html and a servlet whose service method is overridden with code that outputs some HTML tags to the user. This servlet modifies the example below retrieves various information about the user. Instead of sending the information to the console, the service method sends it back to the user.

index2.html

```
<HTML>
<HEAD>
<TITLE>Sending a request</TITLE>
</HEAD>
<BODY>
```

```
<FORM ACTION=
http://localhost:8080/examples/servlets/servlet/RequestDemoServlet
METHOD="POST">
<BR><BR>
Author: <INPUT TYPE="TEXT" NAME="Author">
<INPUT TYPE="SUBMIT" NAME="Submit">
<INPUT TYPE="RESET" VALUE="Reset">
</FORM>
</BODY>
</HTML>
```

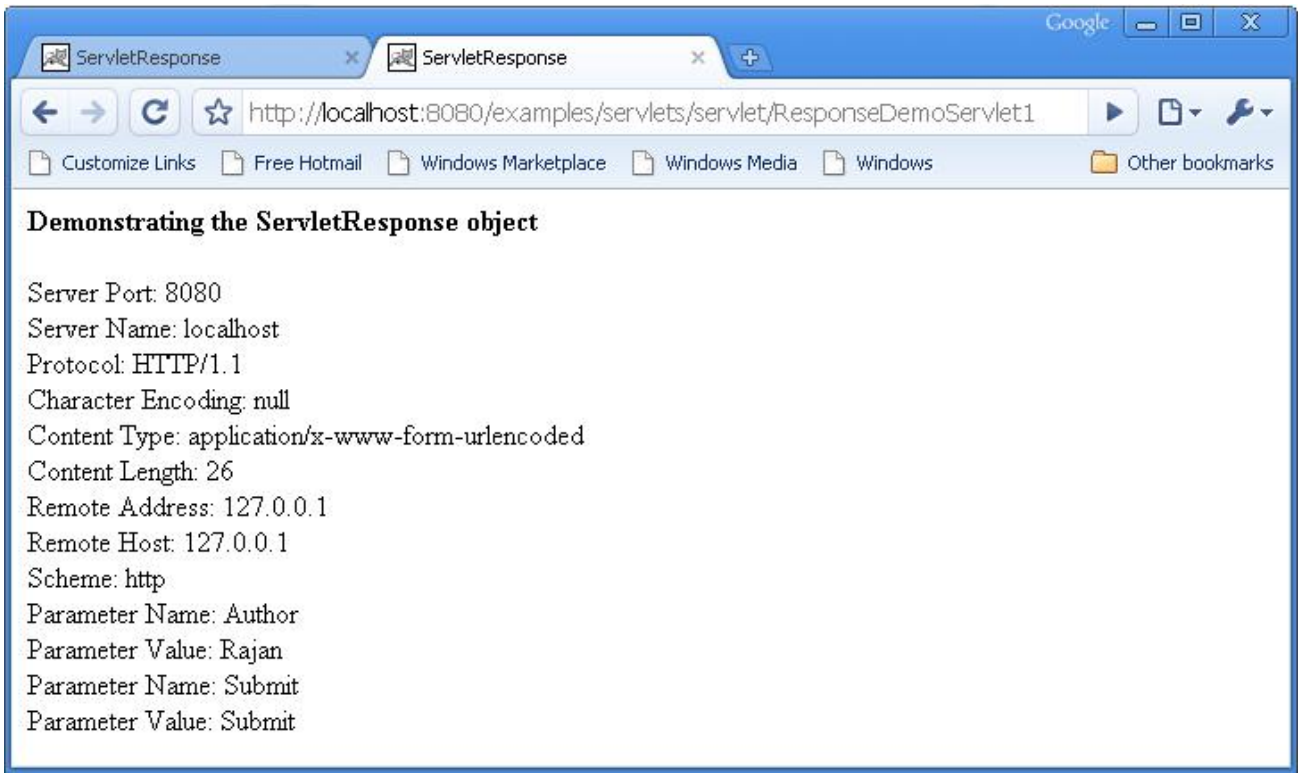
ResponseDemoServlet.java

```
import javax.servlet.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.util.Enumeration;

public class ResponseDemoServlet implements Servlet {
  public void init(ServletConfig config) throws ServletException {
  }
  public void destroy() {
  }
  public void service(ServletRequest request,
                      ServletResponse response)
    throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>");
    out.println("ServletResponse");
    out.println("</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("<B>Demonstrating the ServletResponse object</B>");
    out.println("<BR>");
    out.println("<BR>Server Port: " + request.getServerPort());
    out.println("<BR>Server Name: " + request.getServerName());
    out.println("<BR>Protocol: " + request.getProtocol());
    out.println("<BR>Character Encoding: " +
                request.getCharacterEncoding());
    out.println("<BR>Content Type: " + request.getContentType());
    out.println("<BR>Content Length: " +
                request.getContentLength());
    out.println("<BR>Remote Address: " + request.getRemoteAddr());
    out.println("<BR>Remote Host: " + request.getRemoteHost());
    out.println("<BR>Scheme: " + request.getScheme());
    Enumeration parameters = request.getParameterNames();
    while (parameters.hasMoreElements()) {
```

```
      String parameterName = (String) parameters.nextElement();
      out.println("<br>Parameter Name: " + parameterName);
      out.println("<br>Parameter Value: " +
        request.getParameter(parameterName));
    }
    Enumeration attributes = request.getAttributeNames();
    while (attributes.hasMoreElements()) {
      String attribute = (String) attributes.nextElement();
      out.println("<BR>Attribute name: " + attribute);
      out.println("<BR>Attribute value: " +
                  request.getAttribute(attribute));
    }
    out.println("</BODY>");
    out.println("</HTML>");
  }
  public String getServletInfo() {
    return null;
  }
  public ServletConfig getServletConfig() {
    return null;
  }
}
```

Window of index2.html

Window after clicking the ‘submit’ button



GenericServlet [Ref.1]

Till this point, we have been creating servlet classes that implement the javax.servlet.Servlet interface. Everything works fine, but there are two annoying things that we've probably noticed:

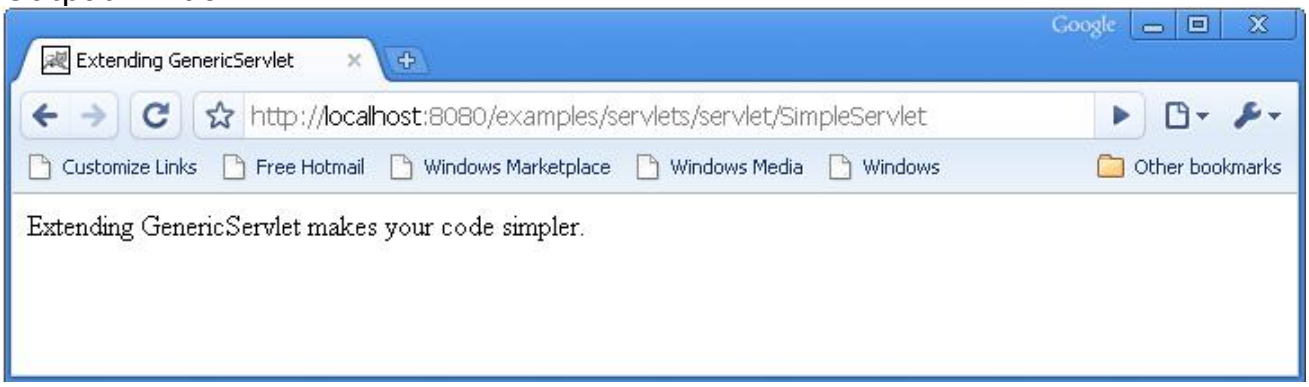
1. We have to provide implementations for all five methods of the Servlet interface, even though most of the time we only need one. This makes your code look unnecessarily complicated.
2. The ServletConfig object is passed to the init method. We need to preserve this object to use it from other methods. This is not difficult, but it means extra work.

The javax.servlet package provides a wrapper class called GenericServlet that implements two important interfaces from the javax.servlet package: Servlet and ServletConfig, as well as the java.io.Serializable interface. The GenericServlet class provides implementations for all methods, most of which are blank. We can extend GenericServlet and override only methods that we need to use. Clearly, this looks like a better solution.

The program given below called SimpleServlet that extends GenericServlet. The code provides the implementation of the service method that sends some output to the browser. Because the service method is the only method we need, only this method needs to appear in the class. Compared to all servlet classes that implement the javax.servlet.Servlet interface directly, SimpleServlet looks much cleaner and clearer.

```
import javax.servlet.*;
import java.io.IOException;
import java.io.PrintWriter;
public class SimpleServlet extends GenericServlet {
  public void service(ServletRequest request,
                      ServletResponse response)
    throws ServletException, IOException {
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>");
    out.println("Extending GenericServlet");
    out.println("</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("Extending GenericServlet makes your code
                          simpler.");
    out.println("</BODY>");
    out.println("</HTML>");
  }
}
```

Output window:



The HttpServlet Class [Ref.1]

The HttpServlet class extends the javax.servlet.GenericServlet class. The HttpServlet class also adds a number of interesting methods to use. The most important are the six doxxx methods that get called when a related HTTP request method is used. The six methods are doPost, doPut, doGet, doDelete, doOptions and doTrace. Each doxxx method is invoked when a corresponding HTTP method is used. For instance, the doGet method is invoked when the servlet receives an HTTP request that was sent using the GET method. Of the six doxxx methods, the doPost and the doGet methods are the most frequently used.

The doPost method is called when the browser sends an HTTP request using the POST method. The POST method is one of the two methods that can be used by an HTML form. Consider the following HTML form at the client side:

```
<FORM ACTION="Register" METHOD="POST">
<INPUT TYPE=TEXT Name="firstName">
<INPUT TYPE=TEXT Name="lastName">
<INPUT TYPE=SUBMIT>
</FORM>
```

When the user clicks the Submit button to submit the form, the browser sends an HTTP request to the server using the POST method. The web server then passes this request to the Register servlet and the doPost method of the servlet is invoked. Using the POST method in a form, the parameter name/value pairs of the form are sent in the request body. For example, if we use the preceding form as an example and enter ‘Sunil’ as the value for firstName and ‘Go’ as the value for lastName, we will get the following result in the request body:

```
firstName=Sunil
lastName=Go
```

An HTML form can also use the GET method; however, POST is much more often used with HTML forms.

The doGet method is invoked when an HTTP request is sent using the GET method. GET is the default method in HTTP. When we type a URL, such as www.yahoo.com, our request is sent to Yahoo! using the GET method. If we use the GET method in a form, the parameter name/value pairs are appended to the URL. Therefore, if we have two parameters named firstName and lastName in our form, and the user enters Sunil and Go, respectively, the URL to our servlet will become something like the following:

```
http://yourdomain/myApp/Register?firstName=Sunil&lastName=Go
```

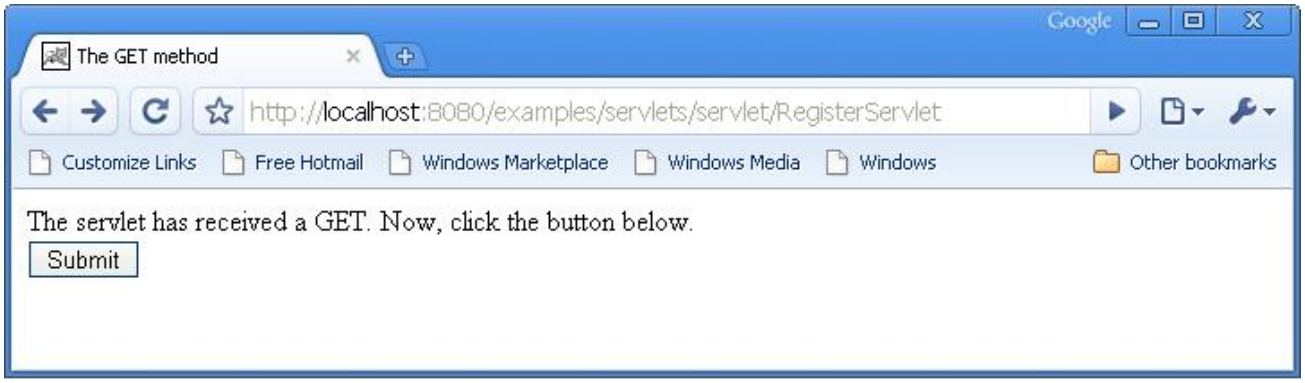
Upon receiving a GET method, the servlet will call its doGet method. The service method of HttpServlet class is as follows:

```
protected void service(HttpServletRequest request,
     HttpServletResponse response)
     throws ServletException, IOException
```

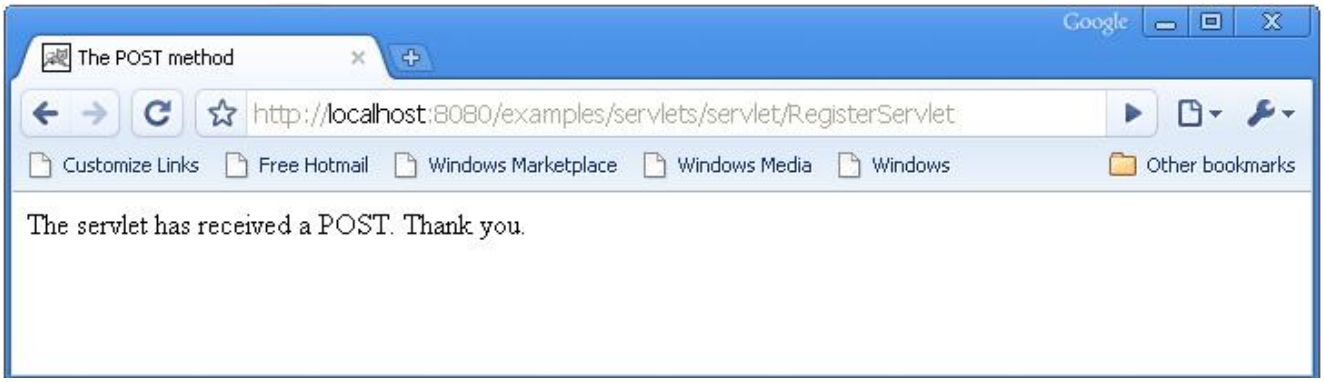
```
//Demonstration of doGet( ) and doPost( ) methods.
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

```
public class RegisterServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>The GET method</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("The servlet has received a GET. " +
      "Now, click the button below.");
    out.println("<BR>");
    out.println("<FORM METHOD=POST>");
    out.println("<INPUT TYPE=SUBMIT VALUE=Submit>");
    out.println("</FORM>");
    out.println("</BODY>");
    out.println("</HTML>");

  }
  public void doPost(
    HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>The POST method</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("The servlet has received a POST. Thank you.");
    out.println("</BODY>");
    out.println("</HTML>");
  }
}
```

When the servlet is first called from a web browser by typing the URL to the servlet in the Address or Location box, GET is used as the request method. At the server side, the doGet method is invoked. The servlet sends a string saying "The servlet has received a GET. Now, click the button below." plus an HTML form. The output is shown in Figure below:



The form sent to the browser uses the POST method. When the user clicks the button to submit the form, a POST request is sent to the server. The servlet then invokes the doPost method, sending a String saying, "The servlet has received a POST. Thank you," to the browser. The output of doPost is shown in Figure below:



HttpServletRequest Interface [Ref.1]

In addition to providing several more protocol-specific methods in the HttpServlet class, the javax.servlet.http package also provides more sophisticated request and response interfaces.

Obtaining HTTP Request Headers from HttpServletRequest [Ref.1]

The HTTP request that a client browser sends to the server includes an HTTP request header with important information, such as cookies and the referer. We can access these headers from the HttpServletRequest object passed to a doxxx method.

The following example demonstrates how we can use the HttpServletRequest interface to obtain all the header names and sends the header name/value pairs to the browser.

```
//Obtaining HTTP request Headers
import javax.servlet.*;
```

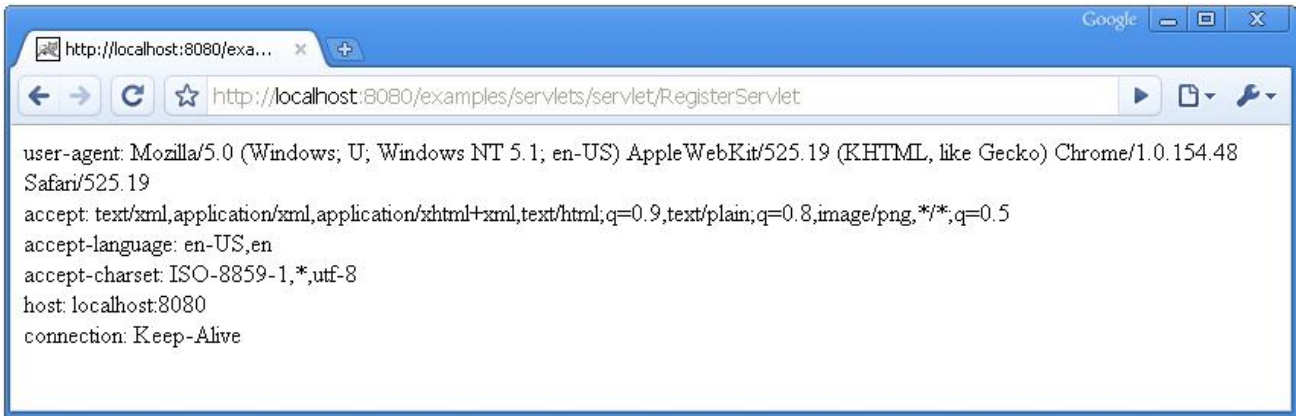
```
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class RegisterServlet extends HttpServlet
{
  public void doGet(HttpServletRequest request,
          HttpServletResponse response)
    throws ServletException, IOException
   {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    Enumeration enumeration = request.getHeaderNames();
    while(enumeration.hasMoreElements())
    {
      String header = (String) enumeration.nextElement();
      out.println(header + ": " + request.getHeader(header) +
                                "<BR>");
    }
  }
}
```

The RegisterServlet given above uses the getHeaderNames and the getHeader methods. The getHeaderNames is first called to obtain an Enumeration containing all the header names found in the client request. The value of each header then is retrieved by using the getHeader method, passing a header name.

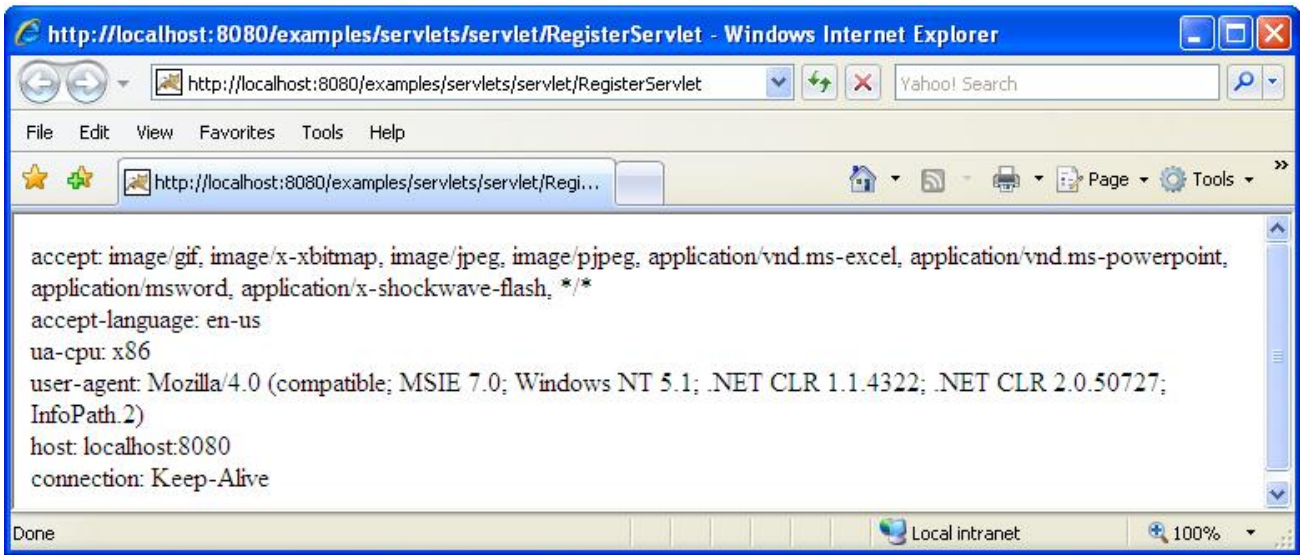
The output of the code depends on the client environment, such as the browser used and the operating system of the client's machine. For example, some browsers might send cookies to the server. Also, whether the servlet is requested by the user typing the URL in the Address/Location box or by clicking a hyperlink also accounts for the presence of an HTTP request header called referer.

The output of the code above is shown in Figure below:

Output obtained in Google Chrome:



Output obtained in Internet Explorer 7:



Obtaining the Query String from HttpServletRequest [Ref.1]

The next important method is the getQueryString method, which is used to retrieve the query string of the HTTP request. A query string is the string on the URL to the right of the path to the servlet.

If we use the GET method in an HTML form, the parameter name/value pairs will be appended to the URL. The code in Listing 3.3 is a servlet named HttpRequestDemoServlet that displays the value of the request's query string and a form.

```
//Obtaining the Query String
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class HttpRequestDemoServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>Obtaining the Query String</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("Query String: " + request.getQueryString() +
                    "<BR>");
    out.println("<FORM METHOD=GET>");
```

```
    out.println("<BR>First Name: <INPUT TYPE=
                    TEXT NAME=FirstName>");
    out.println("<BR>Last Name: <INPUT TYPE=TEXT NAME=LastName>");
    out.println("<BR><INPUT TYPE=SUBMIT VALUE=Submit>");
    out.println("</FORM>");
    out.println("</BODY>");
    out.println("</HTML>");
  }
}
```

When the user enters the URL to the servlet in the web browser and the servlet is first called, the query string is null, as shown in Figure below:

After we enter some values into the HTML form and submit the form, the page is redisplayed. Note that now there is a string added to the URL. The query string has a value of the parameter name/value pairs separated by an ampersand (&). The page is shown in Figure below:

Obtaining the Parameters from HttpServletRequest [Ref.1]

We have seen that we can get the query string containing a value. This means that we can get the form parameter name/value pairs or other values from the previous page. We should not use the getQueryString method to

obtain a form's parameter name/value pairs, however, because this means we have to parse the string ourselves. We can use some other methods in HttpServletRequest to get the parameter names and values: the getParameterNames and the getParameter methods.

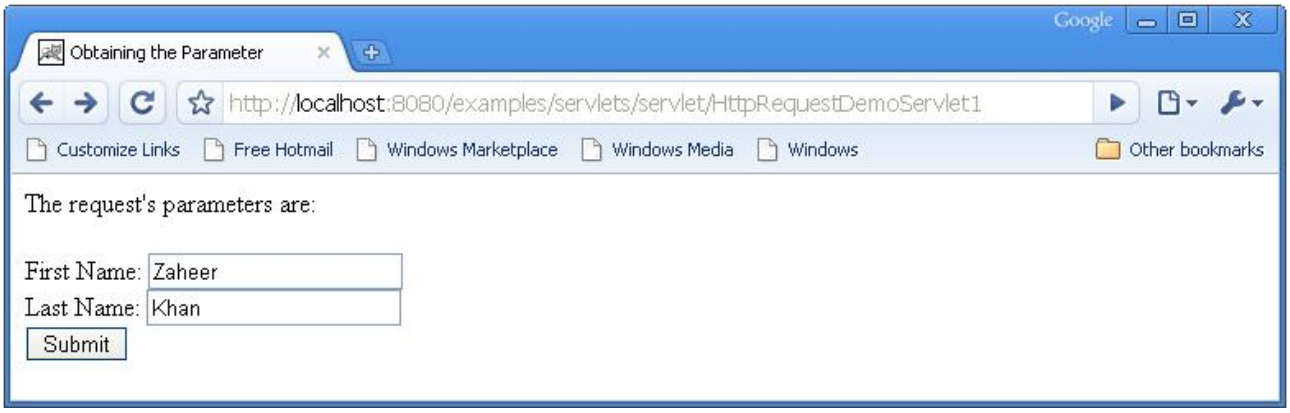
The getParameterNames method returns an Enumeration containing the parameter names. In many cases, however, we already know the parameter names, so we don't need to use this method. To get a parameter value, we use the getParameter method, passing the parameter name as the argument.

The following example demonstrates how we can use the getParameterNames and the getParameter methods to display all the parameter names and values from the HTML form from the previous page. The code is given below:

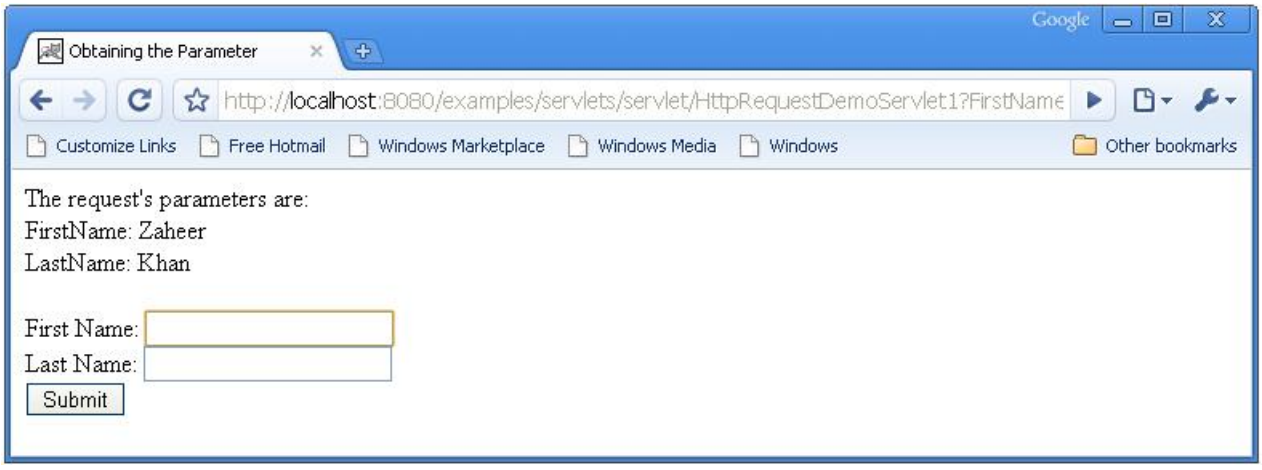
```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class HttpRequestDemoServlet1 extends HttpServlet
{
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
  throws ServletException, IOException
  {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>Obtaining the Parameter</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("The request's parameters are:<BR>");
    Enumeration enumeration = request.getParameterNames();
    while (enumeration.hasMoreElements())
    {
      String parameterName = (String) enumeration.nextElement();
      out.println(parameterName + ": " +
        request.getParameter(parameterName) + "<BR>" );
    }
    out.println("<FORM METHOD=GET>");
    out.println("<BR>First Name: <INPUT TYPE=TEXT
                              NAME=FirstName>");
    out.println("<BR>Last Name: <INPUT TYPE=TEXT NAME=LastName>");
    out.println("<BR><INPUT TYPE=SUBMIT VALUE=Submit>");
    out.println("</FORM>");
    out.println("</BODY>");
    out.println("</HTML>");
  }
```

```
}
```

When the servlet is first called, it does not have any parameter from the previous request. Therefore, the no parameter name/value pair is displayed, as shown in Figure below:



On subsequent requests, the user should enter values for both the firstName and lastName parameters. This is reflected on the next page, which is shown in Figure below:



Manipulating Multi-Value Parameters [Ref.1]

We may have a need to use parameters with the same name in our form. This case might arise, for example, when we are using check box controls that can accept multiple values or when we have a multiple-selection HTML select control. In situations like these, we can't use the getParameter method because it will give us only the first value. Instead, we use the getParameterValues method.

The getParameterValues method accepts one argument: the parameter name. It returns an array of string containing all the values for that parameter.

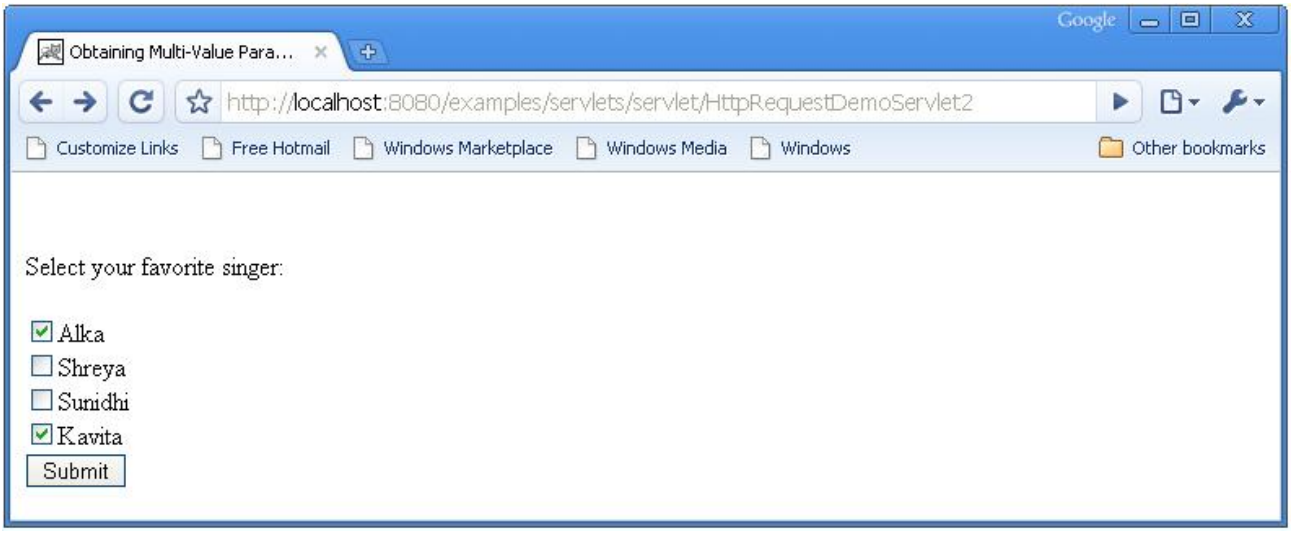
If the parameter of that name is not found, the getParameterValues method will return a null.

The following example illustrates the use of the getParameterValues method to get all favorite music selected by the user. The code for this servlet is given in program below:

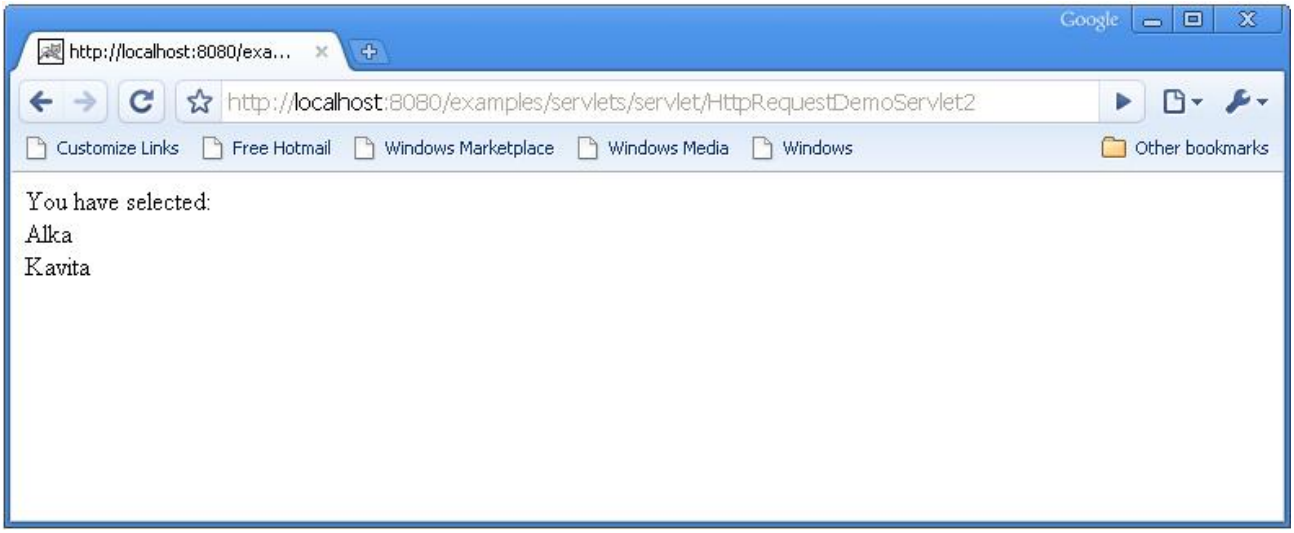
```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class HttpRequestDemoServlet2 extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
  throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>Obtaining Multi-Value Parameters</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("<BR>");
    out.println("<BR>Select your favorite singer:");
    out.println("<BR><FORM METHOD=POST>");
    out.println("<BR><INPUT TYPE=CHECKBOX " +
      "NAME=favoriteMusic VALUE=Alka>Alka");
    out.println("<BR><INPUT TYPE=CHECKBOX " +
      "NAME=favoriteMusic VALUE=Shreya>Shreya");
    out.println("<BR><INPUT TYPE=CHECKBOX " +
      "NAME=favoriteMusic VALUE=Sunidhi>Sunidhi");
    out.println("<BR><INPUT TYPE=CHECKBOX " +
      "NAME=favoriteMusic VALUE=Kavita>Kavita");
    out.println("<BR><INPUT TYPE=SUBMIT VALUE=Submit>");
    out.println("</FORM>");
    out.println("</BODY>");
    out.println("</HTML>");
  }
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
    throws ServletException, IOException {
    String[] values = request.getParameterValues("favoriteMusic");
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    if (values != null ) {
      int length = values.length;
      out.println("You have selected: ");
      for (int i=0; i<length; i++) {
        out.println("<BR>" + values[i]);
```

```
        }
      }
    }
}
```

When the servlet is first called, the doGet method is invoked and the method sends a form to the web browser. The form has four check box controls with the same name: favoriteMusic. Their values are different, however. This is shown in Figure below:



When the user selects the value(s) of the check boxes, the browser sends all selected values. In the server side, we use the getParameterValues to retrieve all values sent in the request. This is shown in Figure below:



Note that we use the POST method for the form; therefore, the parameter name/value pairs are retrieved in the doPost method.

HttpServletResponse [Ref.1]

The HttpServletResponse interface provides several protocol-specific methods not available in the javax.servlet.ServletResponse interface.

The HttpServletResponse interface extends the ServletResponse interface. Till in the examples, we have seen that we always use two of the methods in HttpServletResponse when sending output to the browser: setContentType and getWriter.

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
```

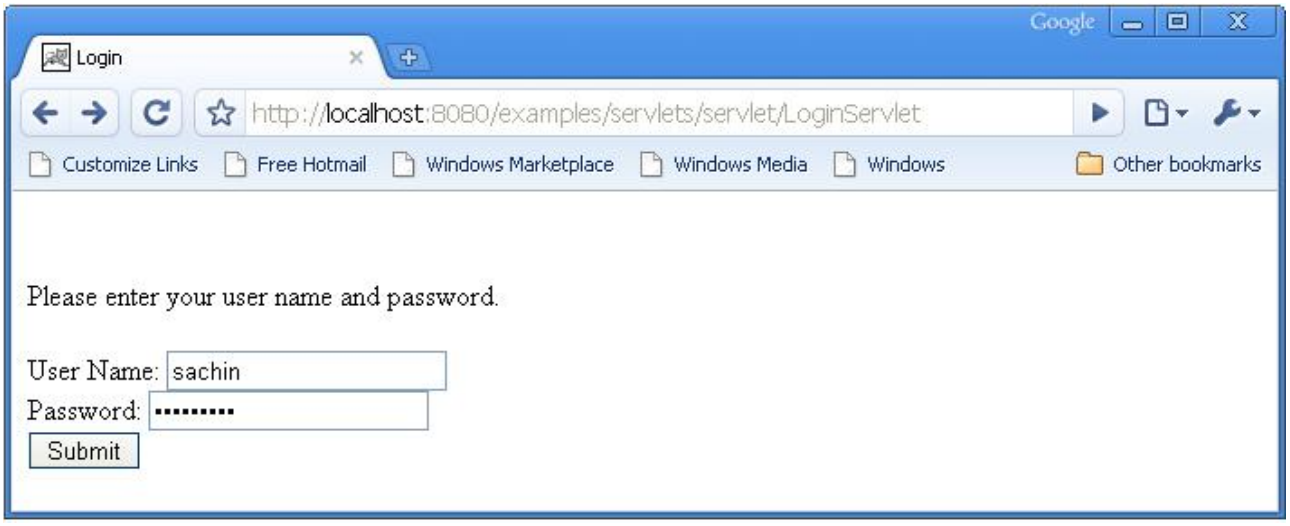
There is more to it, however. The addCookie method sends cookies to the browser. We also use methods to manipulate the URLs sent to the browser. Another interesting method in the HttpServletResponse interface is the setHeader method. This method allows us to add a name/value field to the response header.

We can also use a method to redirect the user to another page: sendRedirect. When we call this method, the web server sends a special message to the browser to request another page. Therefore, there is always a round trip to the client side before the other page is fetched. This method is used frequently and its use is illustrated in the following example. The example below shows a Login page that prompts the user to enter a user name and a password. If both are correct, the user will be redirected to a Welcome page. If not, the user will see the same Login page.

When the servlet is first requested, the servlet's doGet method is called. The doGet method then outputs the form. The user can then enter the user name and password, and submit the form. Note that the form uses the POST method, which means that at the server side, the doPost method is invoked, and the user name and password are checked against some predefined values. If the user name and password match, the user is redirected to a Welcome page. If not, the doPost method outputs the Login form again along with an error message.

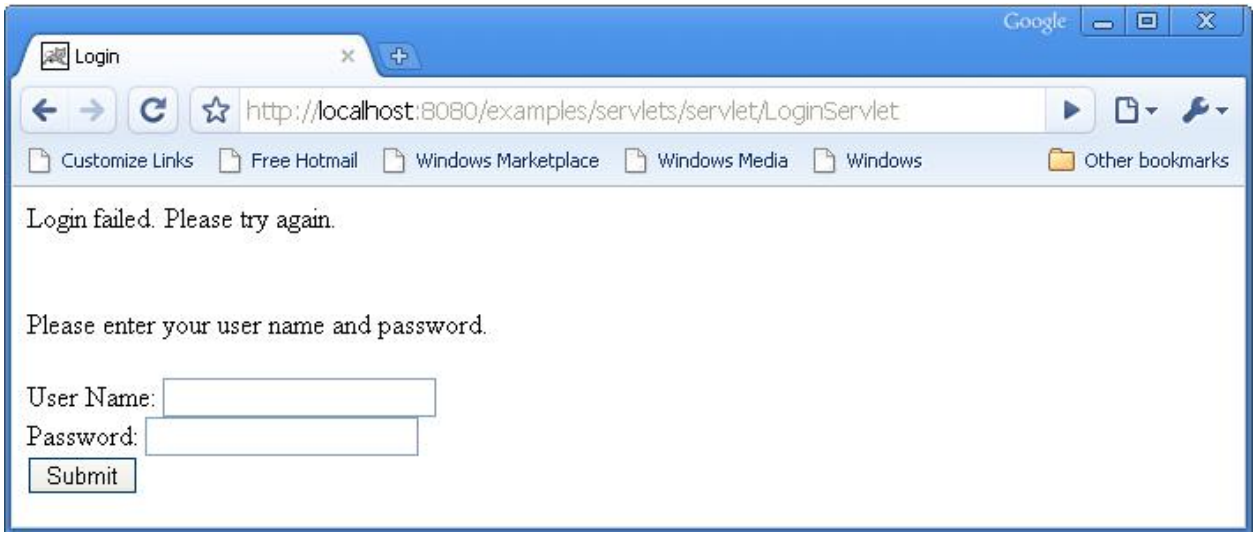
```
public class LoginServlet extends HttpServlet {
  private void sendLoginForm(HttpServletResponse response,
    boolean withErrorMessage)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>Login</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
```

```
    if (withErrorMessage)
      out.println("Login failed. Please try again.<BR>");

    out.println("<BR>");
    out.println("<BR>Please enter your user name and password.");
    out.println("<BR><FORM METHOD=POST>");
    out.println("<BR>User Name: <INPUT TYPE=TEXT NAME=userName>");
    out.println("<BR>Password: <INPUT TYPE=PASSWORD
                    NAME=password>");
    out.println("<BR><INPUT TYPE=SUBMIT VALUE=Submit>");
    out.println("</FORM>");
    out.println("</BODY>");
    out.println("</HTML>");
  }
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    sendLoginForm(response, false);
  }
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
    throws ServletException, IOException {
    String userName = request.getParameter("userName");
    String password = request.getParameter("password");
    if (userName!=null && password!=null &&
      userName.equals("james.bond") && password.equals("007")) {
      response.sendRedirect("http://domain/app/WelcomePage");
    }
    else
      sendLoginForm(response, true);
  }
}
```



In the code given above, private method called sendLoginForm that accepts an HttpServletResponse object and a boolean that signals whether an error message be sent along with the form. This sendLoginForm method is called both from the doGet and the doPost methods. When called from the doGet method, no error message is given, because this is the first time the user requests the page. The withErrorMessage flag is therefore false. When called from the doPost method, this flag is set to true because the sendLoginForm method is only invoked from doPost if the user name and password did not match.

The Login page, when it is first requested, is shown in Figure above. The Login page, after a failed attempt to log in, is shown in Figure below:



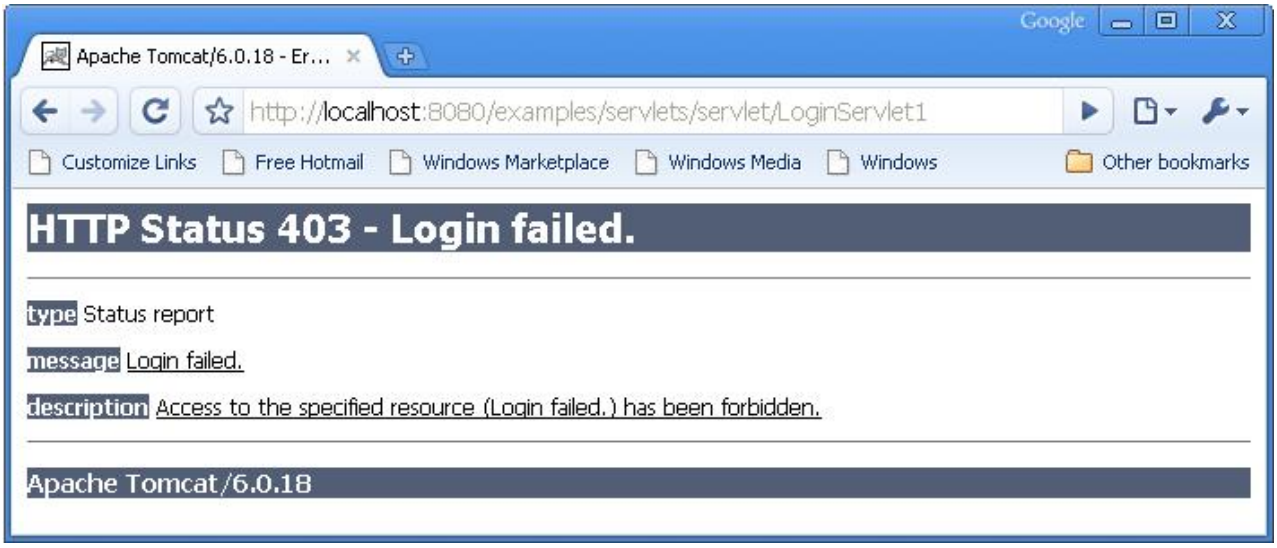
Sending an Error Code [Ref.1]

The HttpServletResponse also allows us to send pre-defined error messages. The interface defines a number of public static final integers that all start with SC_. For example, SC_FORBIDDEN will be translated into an HTTP error 403.

Along with the error code, we also can send a custom error message. Instead of redisplaying the Login page when a failed login occurs, we can send an HTTP error 403 plus our error message. To do this, replace the call to the sendLoginForm in the doPost method with the following:

```
response.sendError(response.SC_FORBIDDEN, "Login failed.");
```

The user will see the screen in following Figure when a login fails.



Request Dispatching [Ref.1]

In some circumstances, we may want to include the content from an HTML page or the output from another servlet. Additionally, there are cases that require that we pass the processing of an HTTP request from our servlet to another servlet. The current servlet specification responds to these needs with an interface called RequestDispatcher, which is found in the javax.servlet package. This interface has two methods, which allow you to delegate the request-response processing to another resource: include and forward. Both methods accept a ServletRequest object and a ServletResponse object as arguments.

As the name implies, the include method is used to include content from another resource, such as another servlet, a JSP page, or an HTML page. The method has the following signature:

```
public void include(javax.servlet.ServletRequest request,
  javax.servlet.ServletResponse response)
  throws javax.servlet.ServletException, java.io.IOException
```

The forward method is used to forward a request from one servlet to another. The original servlet can perform some initial tasks on the ServletRequest object before forwarding it. The signature of the forward method is as follows:

```
public void forward(javax.servlet.ServletRequest request,
  javax.servlet.ServletResponse response)
  throws javax.servlet.ServletException, java.io.IOException
```

The Difference Between sendRedirect and forward:

The sendRedirect method works by sending a status code that tells the browser to request another URL. This means that there is always a round trip to the client side. Additionally, the previous HttpServletRequest object is lost. To pass information between the original servlet and the next request, we normally pass the information as a query string appended to the destination URL.

The forward method, on the other hand, redirects the request without the help from the client's browser. Both the HttpServletRequest object and the HttpServletResponse object also are passed to the new resource.

In order to perform a servlet include or forward, we first need to obtain a RequestDispatcher object. We can obtain a RequestDispatcher object three different ways, as follows:

- Use the getRequestDispatcher method of the ServletContext interface, passing a String containing the path to the other resource. The path is relative to the root of the ServletContext.
- Use the getRequestDispatcher method of the ServletRequest interface, passing a String containing the path to the other resource. The path is relative to the current HTTP request.
- Use the getNamedDispatcher method of the ServletContext interface, passing a String containing the name of the other resource.

Including Static Content [Ref.1]

Sometimes we need to include static content, such as HTML pages or image files that are prepared by a web graphic designer. We can do this by using the same technique for including dynamic resources.

The following example shows a servlet named FirstServlet that includes an HTML file named main.html. The servlet class file is located in the WEB-INF\classes directory, whereas the AdBanner.html file, like other HTML files, resides in the \examples directory. The servlet is given in program below and the HTML file is also given.

```
//Including Static Content
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class FirstServlet extends HttpServlet
{
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    RequestDispatcher rd
                    = request.getRequestDispatcher("/main.html");
    rd.include(request, response);
  }
```

```
}
```

```
//main.html File
<HTML>
<HEAD>
<TITLE>Banner</TITLE>
</HEAD>
<BODY>
<IMG SRC=quote01.jpg>
</BODY>
</HTML>
```

Including another Servlet [Ref.1]

The second example shows a servlet (FirstServlet) that includes another servlet (SecondServlet). The second servlet simply sends the included request parameter to the user. The FirstServlet and the SecondServlet is presented below.

```
//FirstServlet
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class FirstServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>Included Request Parameters</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("<B>Included Request Parameters</B><BR>");
    RequestDispatcher rd =
  request.getRequestDispatcher("/servlet/SecondServlet?name=budi");
    rd.include(request, response);
    out.println("</BODY>");
    out.println("</HTML>");
  }
}
```

```
//SecondServlet
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
```

```
import java.util.*;
public class SecondServlet extends HttpServlet
{
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    Enumeration enum = request.getAttributeNames();
    while (enum.hasMoreElements()) {
      String attributeName = (String) enum.nextElement();
      out.println(attributeName + ": " +
        request.getAttribute(attributeName) + "<BR>");
    }
  }
}
```

Session Tracking and Management [Ref.1]

The Hypertext Transfer Protocol (HTTP) is the network protocol that web servers and client browsers use to communicate with each other. HTTP is the language of the web. HTTP connections are initiated by a client browser that sends an HTTP request. The web server then responds with an HTTP response and closes the connection. If the same client requests another resource from the server, it must open another HTTP connection to the server. The server always closes the connection as soon as it sends the response, whether or not the browser user needs some other resource from the server.

This process is similar to a telephone conversation in which the receiver always hangs up after responding to the last remark/question from the caller. For example, a call goes something like this:

Caller dials. Caller gets connected.

Caller: "Hi, good morning."

Receiver: "Good morning."

Receiver hangs up.

Caller dials again. Caller gets connected.

Caller: "May I speak to Dr. Divakar, please?"

Receiver: "Sure."

Receiver hangs up.

Caller dials again, and so on, and so on.

Putting this in a web perspective, because the web server always disconnects after it responds to a request, the web server does not know whether a request comes from a user who has just requested the first page or from a user who has requested nine other pages before. As such, HTTP is said to be *stateless*.

Being stateless has huge implications. Consider, for example, a user who is shopping at an online store. As usual, the process starts with the user searching for a product. If the product is found, the user then enters the quantity of that product into the shopping cart form and submits it to the server. But, the user is not yet checking out—he still wants to buy something else. So he searches the catalog again for the second product. The first product order has now been lost, however, because the previous connection was closed and the web server does not remember anything about the previous connection.

The good news is that web programmers can work around this. The solution is called user session management. The web server is forced to associate HTTP requests and client browsers. There are four different ways of session tracking:

- User Authentication
- Hidden from fields
- URL Re-writing
- Persistent cookies

User Authentication [Ref.2]

One way to perform session tracking is to leverage the information that comes with user authentication. It occurs when a web server restricts access to some of its resources to only those clients that log in using a recognized username and password. After the client logs in, the username is available to a servlet through getRemoteUser( ).

We can use the username to track a client session. Once a user has logged in, the browser remembers her username and resends the name and password as the user views new pages on the site. A servlet can identify the user through her username and thereby track his session. For example, if the user adds an item to his virtual shopping cart, that fact can be remembered (in a shared class or external database, perhaps) and used later by another servlet when the user goes to the check-out page.

For example, a servlet that utilizes user authentication might add an item to a user's shopping cart with code like the following:

```
String name = req.getRemoteUser();
if (name == null) {
```

```
  // Explain that the server administrator should protect this page
}
else {
  String[] items = req.getParameterValues("item");
  if (items != null) {
    for (int i = 0; i < items.length; i++) {
      addItemToCart(name, items[i]);
    }
  }
}
```

Another servlet can then retrieve the items from a user's cart with code like this:

```
String name = req.getRemoteUser();
if (name == null) {
  // Explain that the server administrator should protect this page
}
else {
  String[] items = getItemsFromCart(name);
}
```

The biggest advantage of using user authentication to perform session tracking is that *it's easy to implement.* Simply tell the server to protect a set of pages, and use getRemoteUser( ) to identify each client. Another advantage is that *the technique works even when the user accesses our site from different machines*. *It also works even if the user strays from our site or exits his browser before coming back*.

The biggest disadvantage of user authentication is that *it requires each user to register for an account and then log in each time he starts visiting our site.* Most users will tolerate registering and logging in as a necessary evil when they are accessing sensitive information, but it's overkill for simple session tracking. Another downside is that HTTP's basic authentication provides *no logout mechanism*; the user has to exit his browser to log out. A final problem with user authentication is that *a user cannot simultaneously maintain more than one session at the same site*. We clearly need alternative approaches to support anonymous session tracking and to support authenticated session tracking with logout.

Hidden Form Fields [Ref.2]

One way to support anonymous session tracking is to use hidden form fields. As the name implies, these are fields added to an HTML form that are not displayed in the client's browser. They are sent back to the server when the form that contains them is submitted. You include hidden form files with HTML like this:

```
<FORM ACTION="/servlet/MovieFinder" METHOD="POST">
...
<INPUT TYPE=hidden NAME="zip" VALUE="94040">
<INPUT TYPE=hidden NAME="level" VALUE="expert">
...
</FORM>
```

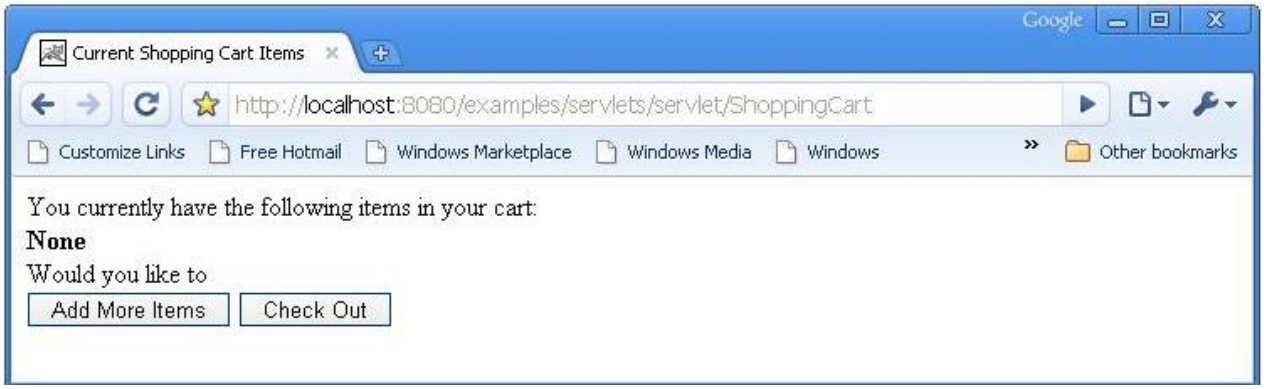
In a sense, hidden form fields define constant variables for a form. To a servlet receiving a submitted form, there is no difference between a hidden field and a visible field.

With hidden form fields, we can rewrite our shopping cart servlets so that users can shop anonymously until checkout time. Example given below demonstrates the technique with a servlet that displays the user's shopping cart contents and lets the user choose to add more items or check out.

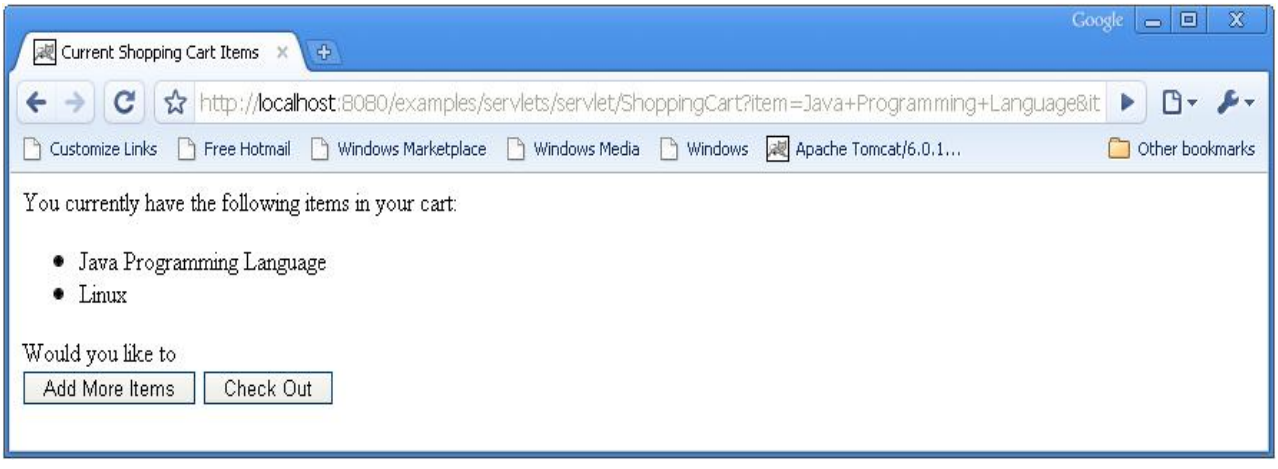
```
//Session Tracking Using Hidden Form Fields
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ShoppingCart extends HttpServlet {
  public void doGet(HttpServletRequest req,
                    HttpServletResponse res) throws
                ServletException, IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();

    out.println("<HEAD><TITLE>Current Shopping Cart
                Items</TITLE></HEAD><BODY>");

    // Cart items are passed in as the item parameter.
    String[] items = req.getParameterValues("item");

    // Print the current cart items.
    out.println("You currently have the following items in
                    your cart:<BR>");
    if (items == null)
      out.println("<B>None</B>");
    else
    {
      out.println("<UL>");
      for (int i = 0; i < items.length; i++)
        out.println("<LI>" + items[i]);
      out.println("</UL>");
    }

    // Ask if the user wants to add more items or check out.
    // Include the current items as hidden fields so they'll
```

```
    // be passed on.
    out.println("<FORM ACTION =
     \"/examples/servlets/servlet/ShoppingCart\" METHOD = POST>");
    if (items != null) {
      for (int i = 0; i < items.length; i++)
        out.println("<INPUT TYPE=HIDDEN NAME=\"item\" VALUE=\"" +
          items[i] + "\">");
    }
    out.println("Would you like to<BR>");
    out.println("<INPUT TYPE=SUBMIT VALUE=\" Add More Items \">");
    out.println("<INPUT TYPE=SUBMIT VALUE=\" Check Out \">");
    out.println("</FORM></BODY></HTML>");
  }
}
```

First view of the window:



After entering the URL as:
http://localhost:8080/examples/servlets/servlet/ShoppingCart?item=Java+Programming+Language&item=Linux



This servlet first reads the items that are already in the cart using getParameterValues("item"). Presumably, the item parameter values were sent

to this servlet using hidden fields. The servlet then displays the current items to the user and asks if he wants to add more items or check out. The servlet asks its question with a form that includes hidden fields, so the form's target (the ShoppingCart servlet) receives the current items as part of the submission.

As more and more information is associated with a client's session, it can become burdensome to pass it all using hidden form fields. In these situations, it's possible to pass on just a unique session ID that identifies a particular client's session. That session ID can be associated with complete information about the session that is stored on the server.

Beware that session IDs must be held as a server secret because any client with knowledge of another client's session ID can, with a forged hidden form field, assume the second client's identity. Consequently, session IDs should be generated so as to be difficult to guess or forge, and active session IDs should be protected—for example, don't make public the server's access log because the logged URLs may contain session IDs for forms submitted with GET requests.

Hidden form fields *can be used to implement authentication with logout.* Simply present an HTML form as the logon screen, and once the user has been authenticated by the server her identity can be associated with her particular session ID. On logout the session ID can be deleted (by not sending the ID to the client on later forms), or the association between ID and user can simply be forgotten.

The advantages of hidden form fields are *their ubiquity and support for anonymity*. Hidden fields are *supported in all the popular browsers, they demand no special server requirements, and they can be used with clients that haven't registered or logged in*. The major disadvantage with this technique, however, is that the session persists only through sequences of dynamically generated forms. *The session cannot be maintained with static documents, emailed documents, bookmarked documents, or browser shutdowns*.

URL Rewriting [Ref.2]

URL rewriting is another way to support anonymous session tracking. With URL rewriting, every local URL the user might click on is dynamically modified, or rewritten, to include extra information. The extra information can be in the form of extra path information, added parameters, or some custom, server-specific URL change. Due to the limited space available in rewriting a URL, the extra information is usually limited to a unique session ID. For example, the following URLs have been rewritten to pass the session ID 123.

http://server:port/servlet/Rewritten	original
http://server:port/servlet/Rewritten/123	extra path information
http://server:port/servlet/Rewritten?sessionid=123	added parameter
http://server:port/servlet/Rewritten;jsessionid=123	custom change

Each rewriting technique has its advantages and disadvantages. *Using extra path information works on all servers, but it doesn't work well if a servlet has to use the extra path information as true path information*. Using an added parameter works on all servers too, but *it can cause parameter naming collisions*. Using a custom, server-specific change works under all conditions for servers that support the change. Unfortunately, it doesn't work at all for servers that don't support the change.

Cookies [Ref.1]

The fourth technique that we can use to manage user sessions is by using cookies. A cookie is a small piece of information that is passed back and forth in the HTTP request and response. Even though a cookie can be created on the client side using some scripting language such as JavaScript, it is usually created by a server resource, such as a servlet. The cookie sent by a servlet to the client will be passed back to the server when the client requests another page from the same application.

Cookies were first specified by Netscape (see http://home.netscape.com/newsref/std/cookie_spec.html) and are now part of the Internet standard as specified in RFC 2109: The HTTP State Management Mechanism. Cookies are transferred to and from the client in the HTTP headers.

In servlet programming, a cookie is represented by the Cookie class in the javax.servlet.http package. We can create a cookie by calling the Cookie class constructor and passing two String objects: the name and value of the cookie. For instance, the following code creates a cookie object called c1. The cookie has the name "myCookie" and a value of "secret":

```
Cookie c1 = new Cookie("myCookie", "secret");
```

We then can add the cookie to the HTTP response using the addCookie method of the HttpServletResponse interface:

```
response.addCookie(c1);
```

Note that because cookies are carried in the request and response headers, we must not add a cookie after an output has been written to the HttpServletResponse object. Otherwise, an exception will be thrown.

The following example shows how we can create two cookies called userName and password and illustrates how those cookies are transferred back to the server. The servlet is called CookieServlet, and its code is given in program below:

When it is first invoked, the doGet method of the servlet is called. The method creates two cookies and adds both to the HttpServletResponse object, as follows:

```
Cookie c1 = new Cookie("userName", "Helen");
Cookie c2 = new Cookie("password", "Keppler");
response.addCookie(c1);
response.addCookie(c2);
```

Next, the doGet method sends an HTML form that the user can click to send another request to the servlet:

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<HTML>");
out.println("<HEAD>");
out.println("<TITLE>Cookie Test</TITLE>");
out.println("</HEAD>");
out.println("<BODY>");
out.println("Please click the button to see
                the cookies sent to you.");
out.println("<BR>");
out.println("<FORM METHOD=POST>");
out.println("<INPUT TYPE=SUBMIT VALUE=Submit>");
out.println("</FORM>");
out.println("</BODY>");
out.println("</HTML>");
```

The form does not have any element other than a submit button. When the form is submitted, the doPost method is invoked. The doPost method does two things: It iterates all the headers in the request to show how the cookies are conveyed back to the server, and it retrieves the cookies and displays their values.

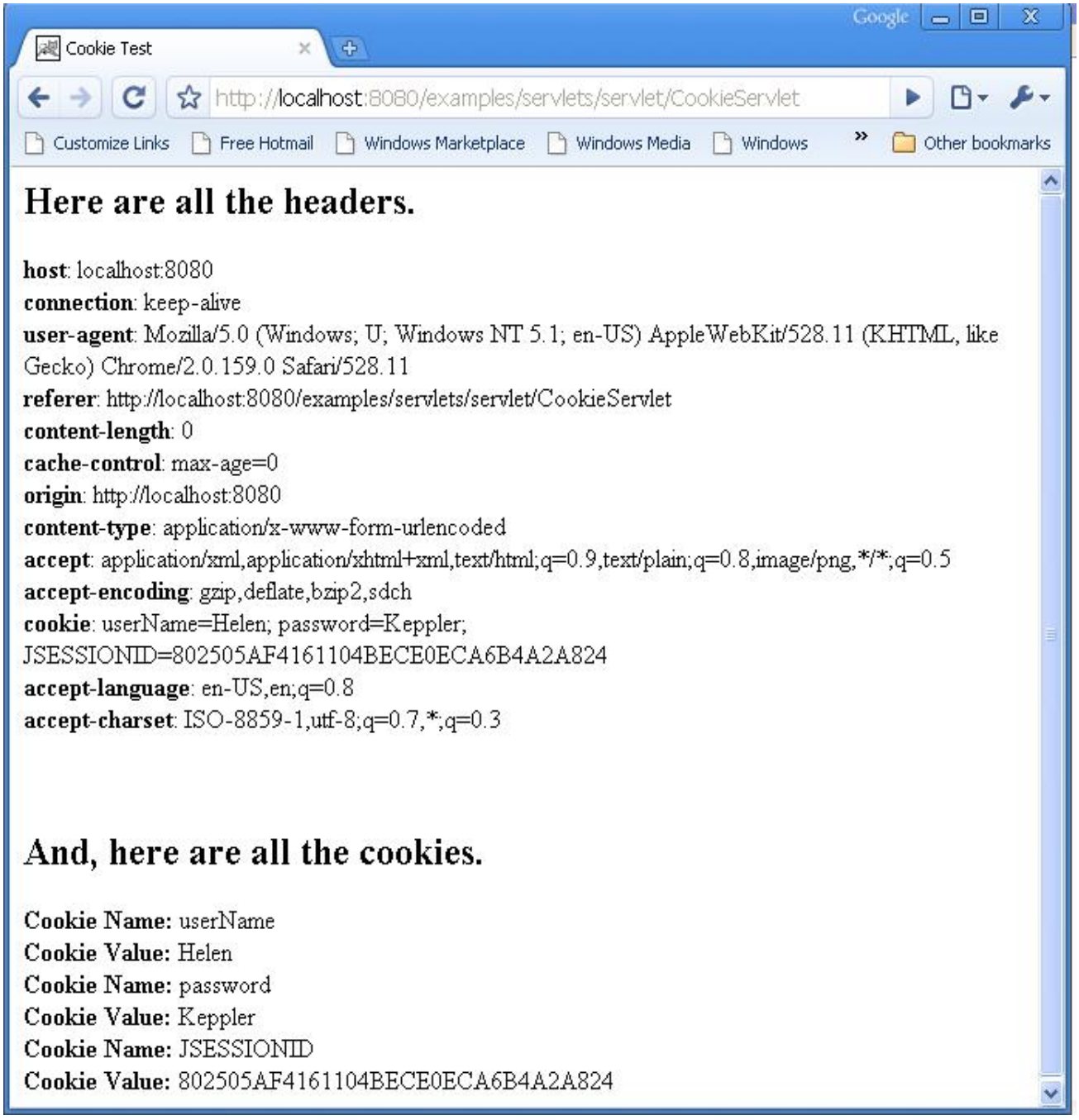
For displaying all the headers in the HttpServletRequest method, it first retrieves an Enumeration object containing all the header names. The method then iterates the Enumeration object to get the next header name and passes the header name to the getHeader method to display the value of that header, as we see here:

```
Enumeration enumr = request.getHeaderNames();
while (enumr.hasMoreElements())
{
  String header = (String) enumr.nextElement();
  out.print("<B>" + header + "</B>: ");
  out.print(request.getHeader(header) + "<BR>");
}
```

In order to retrieve cookies, we use the getCookies method of the HttpServletRequest interface. This method returns a Cookie array containing all cookies in the request. It is our responsibility to loop through the array to get the cookie we want, as follows:

```
     Cookie[] cookies = request.getCookies();
     int length = cookies.length;
     for (int i=0; i<length; i++)
     {
       Cookie cookie = cookies[i];
       out.println("<B>Cookie Name:</B> " +
         cookie.getName() + "<BR>");
       out.println("<B>Cookie Value:</B> " +
         cookie.getValue() + "<BR>");
     }

//The Cookie Servlet
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
public class CookieServlet extends HttpServlet {
  /**Process the HTTP Get request*/
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    Cookie c1 = new Cookie("userName", "Helen");
    Cookie c2 = new Cookie("password", "Keppler");
    response.addCookie(c1);
    response.addCookie(c2);

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>Cookie Test</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("Please click the button to see
                    the cookies sent to you.");
    out.println("<BR>");
    out.println("<FORM METHOD=POST>");
    out.println("<INPUT TYPE=SUBMIT VALUE=Submit>");
    out.println("</FORM>");
    out.println("</BODY>");
    out.println("</HTML>");
  }
  /**Process the HTTP Post request*/
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response) throws
                         ServletException,IOException {
    response.setContentType("text/html");
```

```
    PrintWriter out = response.getWriter();
    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>Cookie Test</TITLE>");
    out.println("</HEAD>");
    out.println("<BODY>");
    out.println("<H2>Here are all the headers.</H2>");

    Enumeration enumr = request.getHeaderNames();
    while (enumr.hasMoreElements()) {
      String header = (String) enumr.nextElement();
      out.print("<B>" + header + "</B>: ");
      out.print(request.getHeader(header) + "<BR>");
    }
    out.println("<BR><BR><H2>And, here are all the
          cookies.</H2>");
    Cookie[] cookies = request.getCookies();
    int length = cookies.length;
    for (int i=0; i<length; i++)
    {
      Cookie cookie = cookies[i];
      out.println("<B>Cookie Name:</B> " + cookie.getName()
                              + "<BR>");
      out.println("<B>Cookie Value:</B> " + cookie.getValue()
                              + "<BR>");
    }
     out.println("</BODY>");
     out.println("</HTML>");
   }
}
```



Simple Cookie operations

Setting Cookie Attributes:

The Cookie class provides a number of methods for setting a cookie's values and attributes. Using these methods is straightforward. The following example sets the comment field of the Servlet's cookie. The comment field describes the purpose of the cookie.

```
  public void doGet (HttpServletRequest request,
```

```
                    HttpServletResponse response)
 throws ServletException, IOException
{
    ...
    // If the user wants to add a book, remember it
    // by adding a cookie
    if (values != null)
    {
        bookId = values[0];
        Cookie getBook = new Cookie("Buy", bookId);
        getBook.setComment("User wants to buy this book " +
                           "from the bookstore.");
    }
    ...
}
```

We can also set the maximum age of the cookie. This attribute is useful, for example, for deleting a cookie. Once again, if Duke's Bookstore kept track of a user's order with cookies, the example would use this attribute to delete a book from the user's order. The user removes a book from the shopping cart in the Servlet; its code would look something like this:

```
public void doGet (HttpServletRequest request,
                   HttpServletResponse response)
 throws ServletException, IOException
{
 ...
    /* Handle any pending deletes from the shopping cart */
    String bookId = request.getParameter("Remove");
    ...
    if (bookId != null) {
      // Find the cookie that pertains to the book to remove
        ...
      // Delete the cookie by setting its maximum age to zero
        thisCookie.setMaxAge(0);
        ...
    }
 // also set content type header before accessing the Writer
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    //Print out the response
    out.println("<html> <head>" +
                "<title>Your Shopping Cart</title>" + ...);
```

Sending the Cookie

Cookies are sent as headers of the response to the client; they are added with the addCookie method of the HttpServletResponse class. If we are using a

Writer to return text data to the client, we must call the addCookie method before calling the HttpServletResponse's getWriter method.

The following is code for sending the cookie:

```
public void doGet (HttpServletRequest request,
                   HttpServletResponse response)
 throws ServletException, IOException
{
    ...
  //If user wants to add a book, remember it by adding a cookie
    if (values != null) {
        bookId = values[0];
        Cookie getBook = new Cookie("Buy", bookId);
        getBook.setComment("User has indicated a desire " +
                    "to buy this book from the bookstore.");
        response.addCookie(getBook);
    }
    ...
}
```

Retrieving Cookies

Clients return cookies as fields added to HTTP request headers. To retrieve any cookie, we must retrieve all the cookies using the getCookies method of the HttpServletRequest class.

The getCookies method returns an array of Cookie objects, which we can search to find the cookie or cookies that we want. (Remember that multiple cookies can have the same name. In order to get the name of a cookie, use its getName method.)

For example:

```
public void doGet (HttpServletRequest request,
                   HttpServletResponse response)
 throws ServletException, IOException
{
 ...
    /* Handle any pending deletes from the shopping cart */
    String bookId = request.getParameter("Remove");
    ...
    if (bookId != null) {
      // Find the cookie that pertains to the book to remove
        Cookie[] cookies = request.getCookies();
        ...
      // Delete the book's cookie by setting its max age to 0
        thisCookie.setMaxAge(0);
    }
```

```
     // also set content type header before accessing the Writer
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        //Print out the response
        out.println("<html> <head>" +
                    "<title>Your Shopping Cart</title>" + ...);
```

Getting the Value of a Cookie

To find the value of a cookie, use its getValue method. For example:

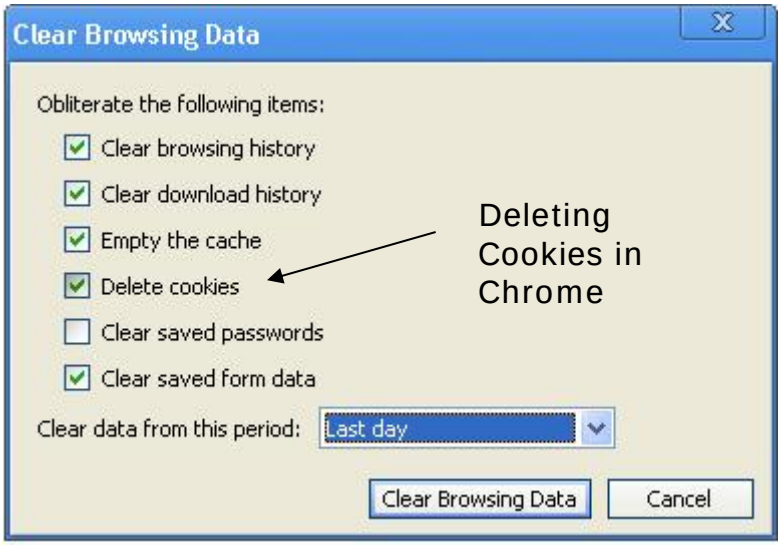
```
    public void doGet (HttpServletRequest request,
                       HttpServletResponse response)
     throws ServletException, IOException
    {
        ...
        /* Handle any pending deletes from the shopping cart */
        String bookId = request.getParameter("Remove");
        ...
        if (bookId != null) {
            // Find the cookie that pertains to that book
            Cookie[] cookies = request.getCookies();
            for(i=0; i < cookies.length; i++) {
                Cookie thisCookie = cookie[i];
                if (thisCookie.getName().equals("Buy") &&
                    thisCookie.getValue().equals(bookId)) {
              // Delete cookie by setting its maximum age to zero
                    thisCookie.setMaxAge(0);
                }
            }
        }
     // also set content type header before accessing the Writer
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        //Print out the response
        out.println("<html> <head>" +
                    "<title>Your Shopping Cart</title>" + ...);
```

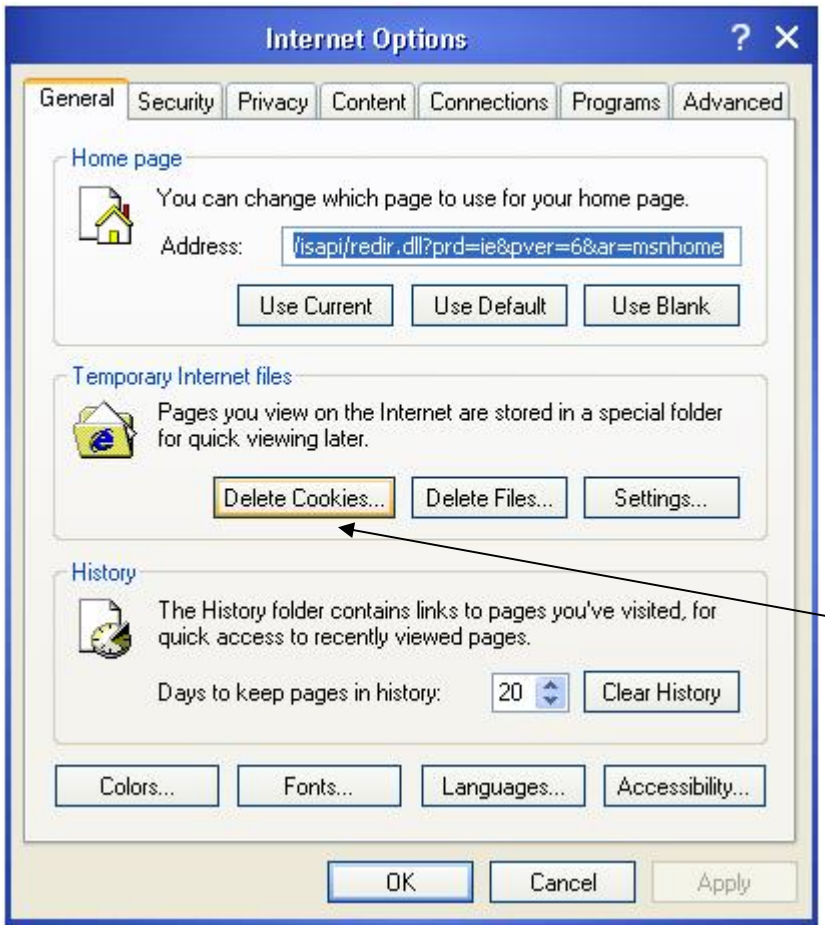
//Example of cookie:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class CookieExample extends HttpServlet {
    public void doGet(   HttpServletRequest request,
                         HttpServletResponse response)
    throws IOException, ServletException
    {
```

```
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        // print out cookies
        Cookie[] cookies = request.getCookies();
        for (int i = 0; i < cookies.length; i++) {
            Cookie c = cookies[i];
            String name = c.getName();
            String value = c.getValue();
            out.println(name + " = " + value);
        }
        // set a cookie
        String name = request.getParameter("cookieName");
        if (name != null && name.length() > 0) {
            String value = request.getParameter("cookieValue");
            Cookie c = new Cookie(name, value);
            response.addCookie(c);
        }
    }
}
```

For deleting the persistent cookies, use following windows of Google Chrome and Internet Explorer respectively:





Deleting Cookies in internet Explorer

Session Objects [Ref.1]

Out of the four techniques for session management, the Session object, represented by the javax.servlet.http.HttpSession interface, is the easiest to use and the most powerful. For each user, the servlet can create an HttpSession object that is associated with that user only and can only be accessed by that particular user. The HttpSession object acts like a Hashtable into which we can store any number of key/object pairs. The HttpSession object is accessible from other servlets in the same application. To retrieve an object previously stored, we need only to pass the key.

An HttpSession object uses a cookie or URL rewriting to send a token to the client. If cookies are used to convey session identifiers, the client browsers are required to accept cookies.

Unlike previous techniques, however, the server does not send any value. What it sends is simply a unique number called the session identifier. This session identifier is used to associate a user with a Session object in the server. Therefore, if there are 10 simultaneous users, 10 Session objects will be created in the server and each user can access only his/her own HttpSession object.

The way an HttpSession object is created for a user and retrieved in the next requests is illustrated in Figure below:

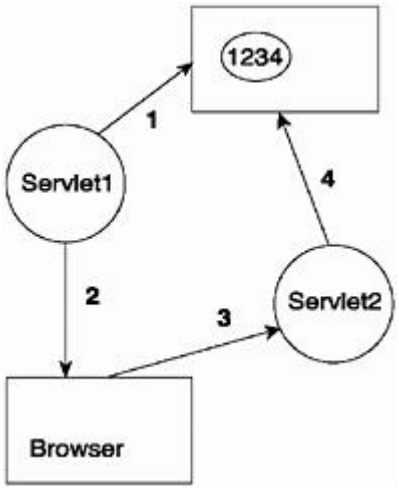


Figure above shows that there are four steps in session tracking using the HttpSession object:

1. An HttpSession object is created by a servlet called Servlet1. A session identifier is generated for this HttpSession object. In this example, the session identifier is 1234, but in reality, the servlet container will generate a longer random number that is guaranteed to be unique. The HttpSession object then is stored in the server and is associated with the generated session identifier. Also the programmer can store values immediately after creating an HttpSession.

2. In the response, the servlet sends the session identifier to the client browser.

3. When the client browser requests another resource in the same application, such as Servlet2, the session identifier is sent back to the server and passed to Servlet2 in the HttpServletRequest object.

4. For Servlet2 to have access to the HttpSession object for this particular client, it uses the getSession method of the HttpServletRequest interface. This method automatically retrieves the session identifier from the request and obtains the HttpSession object associated with the session identifier.

The getSession method of the HttpServletRequest interface has two overloads. They are as follows:

```
HttpSession getSession()
```

```
HttpSession getSession(boolean create)
```

The first overload returns the current session associated with this request, or if the request does not have a session identifier, it creates a new one.

The second overload returns the HttpSession object associated with this request if there is a valid session identifier in the request. If no valid session identifier is found in the request, whether a new HttpSession object is created depends on the create value. If the value is true, a new HttpSession object is created if no valid session identifier is found in the request. Otherwise, the getSession method will return null.

The HttpSession interface [Ref.1]

getAttribute

This method retrieves an attribute from the HttpSession object. The return value is an object of type Object; therefore we may have to downcast the attribute to its original type. To retrieve an attribute, we pass the name associated with the attribute. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object.
The signature:

```
public Object getAttribute(String name)
     throws IllegalStateException
```

getAttributeNames

The getAttributeNames method returns a java.util.Enumeration containing all attribute names in the HttpSession object. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object.
The signature is as follows:

```
public java.util.Enumeration getAttributeNames()
     throws IllegalStateException
```

getCreationTime

The getCreationTime method returns the time that the HttpSession object was created, in milliseconds since January 1, 1970 00:00:00 GMT. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object.
The signature is as follows:

```
public long getCreationTime() throws IllegalStateException
```

getId

The getID method returns the session identifier. The signature for this method is as follows:

```
public String getId()
```

getLastAccessedTime

The getLastAccessedTime method returns the time the HttpSession object was last accessed by the client. The return value is the number of milliseconds lapsed since January 1, 1970 00:00:00 GMT. The following is the method signature:

```
public long getLastAccessedTime()
```

getMaxInactiveInterval

The getMaxInactiveInterval method returns the number of seconds the HttpSession object will be retained by the servlet container after it is last accessed before being removed. The signature of this method is as follows:

```
public int getMaxInactiveInterval()
```

getServletContext

The getServletContext method returns the ServletContext object the HttpSession object belongs to. The signature is as follows:

```
public javax.servlet.ServletContext getServletContext
```

invalidate

The invalidate method invalidates the HttpSession object and unbinds any object bound to it. This method throws an IllegalStateException if this method is called upon an already invalidated HttpSession object. The signature is as follows:

```
public void invalidate() throws IllegalStateException
```

isNew

The isNew method indicates whether the HttpSession object was created with this request and the client has not yet joined the session tracking. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object. Its signature is as follows:

```
public boolean isNew() throws IllegalStateException
```

removeAttribute

The removeAttribute method removes an attribute bound to this HttpSession object. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object. Its signature is as follows:

```
public void removeAttribute(String name)
     throws IllegalStateException
```

setAttribute

The setAttribute method adds a name/attribute pair to the HttpSession object. This method returns an IllegalStateException if it is called upon an invalidated HttpSession object. The method has the following signature:

```
public void setAttribute(String name, Object attribute)
     throws IllegalStateException
```

setMaxInactiveInterval

The setMaxInactiveInterval method sets the number of seconds from the time the HttpSession object is accessed the servlet container will wait before removing the HttpSession object. The signature is as follows:

```
public void setMaxInactiveInterval(int interval)
```

Passing a negative number to this method will make this HttpSession object never expires.

Example:

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SessionExample extends HttpServlet {
    public void doGet(   HttpServletRequest request,
                         HttpServletResponse response)
    throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        HttpSession session = request.getSession(true);
```

```
        // print session info
        Date created = new Date(session.getCreationTime());
        Date accessed = new Date(session.getLastAccessedTime());
        out.println("ID " + session.getId());
        out.println("Created: " + created);
        out.println("Last Accessed: " + accessed);

        // set session info if needed
        String dataName = request.getParameter("dataName");
        if (dataName != null && dataName.length() > 0) {
            String dataValue = request.getParameter("dataValue");
            session.setAttribute(dataName, dataValue);
        }

        // print session contents
        Enumeration e = session.getAttributeNames();
        while (e.hasMoreElements()) {
            String name = (String)e.nextElement();
            String value = session.getAttribute(name).toString();
            out.println(name + " = " + value);
        }
    }
}
```

References

1. **Java for the Web with Servlets, JSP, and EJB: A Developer's Guide to J2EE Solutions,**
   First Edition by Budi Kurniawan, 2002, New Riders Publishing
   Chapter 1: The Servlet Technology
   Chapter 2: Inside Servlets
   Chapter 3: Writing Servlet Applications
   Chapter 5: Session Management
   (Most of the data is referred from this book)

2. **Java Servlet Programming,**
   Second Edition by Jason Hunter, William Crawford, 2001, O'Reilly

3. **Java the Complete Reference**,
   Seventh Edition by Herbert Schildt, 2001 Osborne McGraw Hill
   Chapter 31: Servlets

Notes

- Most of the books mentioned as reference for the chapters are available in e-copy on the following sites. You can use them for reference too.
  - www.esnips.com
  - www.4shared.com
  - www.isbnonline.com
  - www.pdfchm.com
  - www.scribd.com
- I have referred many of the Power Point Presentations also for reference. I express sincere thanks to the creators of those Power Point Presentations. All these presentations are available on following link:
  http://sites.google.com/site/advancedjavabooksandppts
- The reference material that is not been included in this book such as review questions, practical questions, objective questions, notes, question papers, program's soft copy are available on following link of Google sites:
  http://sites.google.com/site/tusharkute
- The following software are used for compiling programs:
  Java Development Kit Version 6.0 java.sun.com
  Apache Tomcat Server Version 6.0.18
- The following software are used for formatting this documents:
  Editing: Microsoft Office Word 2003, Windows notepad.
  Picture Editing: Windows Paint, Microsoft Office Picture Manager 2003
  Database: Microsoft Office Access 2003
  Web Browsers: Google Chrome Ver 2.0.159, Internet Explorer Ver 6.0
  PDF Creation: Adobe PDF Maker 6.0
- You are free to send the queries and suggestions at:
  tbkute@gmail.com
