REVIEW OF APPLETS:

APPLET:

Applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a Web document.

After an applet arrives on the client, it has limited access to resources, so that it can produce an arbitrary multimedia user interface and run complex computations without introducing the risk of viruses or breaching data integrity.

Let's begin with the simple applet shown here:

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
public void paint(Graphics g) {
g.drawString("A Simple Applet", 20, 20);
}
}
```

This applet begins with two import statements. The first imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user through the AWT, not through the console-based I/O classes. The AWT contains support for a window-based, graphical interface.

The second import statement imports the applet package, which contains the class Applet. Every applet that you create must be a subclass of Applet.

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

Inside SimpleApplet, paint( ) is declared. This method is defined by the AWT and must be overridden by the applet. paint( ) is called each time that the applet must redisplay its output. This situation can occur for several reasons.

For example:

The window in which the applet is running can be overwritten by another window and then uncovered. Or, the applet window can be minimized and then restored. paint( ) is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, paint( ) is called. The paint( ) method has one parameter of type Graphics. This parameter contains the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

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

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

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

Notice that the applet does not have a main( ) method. Unlike Java programs, applets do not begin execution at main( ). In fact, most applets don't even have a main( ) method. Instead, an applet begins execution when the name of its class is passed to an applet viewer or to a network browser.

jkdirectory jkmaterials

After you enter the source code for SimpleApplet, compile in the same way that you have been compiling programs. However, running SimpleApplet involves a different process. In fact, there are two ways in which you can run an applet:

- Executing the applet within a Java-compatible Web browser.
- Using an applet viewer, such as the standard SDK tool, appletviewer. An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.

To execute an applet in a Web browser, you need to write a short HTML text file that contains the appropriate APPLET tag. Here is the HTML file that executes SimpleApplet:

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

The width and height statements specify the dimensions of the display area used by the applet. After you create this file, you can execute your browser and then load this file, which causes SimpleApplet to be executed.

To execute SimpleApplet with an applet viewer, you may also execute the HTML file shown earlier. For example, if the preceding HTML file is called RunApp.html, then the following command line will run SimpleApplet:

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

However, a more convenient method exists that you can use to speed up testing. Simply include a comment at the head of your Java source code file that contains the APPLET tag. By doing so, your code is documented with a prototype of the necessary HTML statements, and you can test your compiled applet merely by starting the applet viewer with your Java source code file. If you use this method, the SimpleApplet source file looks like this:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/
public class SimpleApplet extends Applet {
public void paint(Graphics g) {
g.drawString("A Simple Applet", 20, 20);
}
}
```

In general, you can quickly iterate through applet development by using these three steps:

1. Edit a Java source file.
2. Compile your program.
3. Execute the applet viewer, specifying the name of your applet's source file. The applet viewer will encounter the APPLET tag within the comment and execute your applet.

The window produced by SimpleApplet, as displayed by the applet viewer, is shown below:

APPLET CLASS:

The Applet class is contained in the java.applet package. Applet contains several methods that give you detailed control over the execution of your applet. All applets are subclasses of Applet. Thus, all applets must import java.applet. Applets must also import java.awt. Since all applets run in a window, it is necessary to include support for that window. Applets are not executed by the console-based Java run-time interpreter. Rather, they are executed by either a Web browser or an applet viewer.

Execution of an applet does not begin at main( ). Actually, few applets even have main( ) methods. Instead, execution of an applet is started and controlled with an entirely different mechanism. Output to your applet's window is not performed by System.out.println( ). Rather, it is handled with various AWT methods, such as drawString( ), which outputs a string to a specified X,Y location. Input is also handled differently than in an application.

Once an applet has been compiled, it is included in an HTML file using the APPLET tag. The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTML file. To view and test an applet more conveniently, simply include a comment at the head of your Java source code file that contains the APPLET tag. This way, your code is documented with the necessary HTML statements needed by your applet, and you can test the compiled applet by starting the applet viewer with your Java source code file specified as the target. Here is an example of such a comment:

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

This comment contains an APPLET tag that will run an applet called MyApplet in a window that is 200 pixels wide and 60 pixels high.

THE APPLET CLASS:

The Applet class defines the methods shown in Table given below. Applet provides all necessary support for applet execution, such as starting and stopping. It also provides methods that load and display images, and methods that load and play audio clips. Applet extends the AWT class Panel. In turn, Panel extends Container, which extends Component. These classes provide support for Java's window-based, graphical interface. Thus, Applet provides all of the necessary support for window-based activities.

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

void play(URL url)	If an audio clip is found at the location specified by url, the clip is played.
void play(URL url, String clipName)	If an audio clip is found at the location specified by url with the name specified by clipName, the clip is played.
void resize(Dimension dim)	Resizes the applet according to the dimensions specified by dim. Dimension is a class stored inside java.awt. It contains two integer fields: width and height.
void resize(int width, int height)	Resizes the applet according to the dimensions specified by width and height.
void showStatus(String str)	Displays str in the status window of the browser or applet viewer. If the browser does not support a status window, then no action takes place.
void start( )	Called by the browser when an applet should start (or resume) execution. It is automatically called after init( ) when an applet first begins.
void stop( )	Called by the browser to suspend execution of the applet. Once stopped, an applet is restarted when the browser calls start( ).

AN APPLET SKELETON:

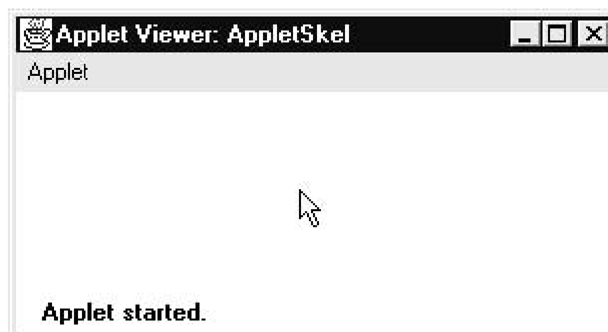
All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods—init( ), start( ), stop( ), and destroy( )—are defined by Applet. Another, paint( ), is defined by the AWT Component class. Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use. However, only very simple applets will not need to define all of them. These five methods can be assembled into the skeleton shown here:

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/
public class AppletSkel extends Applet {
// Called first.
public void init() {
// initialization
}
/* Called second, after init(). Also called whenever
the applet is restarted. */
public void start() {
// start or resume execution
}
// Called when the applet is stopped.
public void stop() {
// suspends execution
}
```

jkmaterials jkdirectory

```
/* Called when applet is terminated. This is the last
method executed. */
public void destroy() {
// perform shutdown activities
}
// Called when an applet's window must be restored.
public void paint(Graphics g) {
// redisplay contents of window
}
}
```

Although this skeleton does not do anything, it can be compiled and run. When run, it generates the following window when viewed with an applet viewer:



APPLET INITIALIZATION AND TERMINATION:

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

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

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

1. stop( )
2. destroy( )

The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

The **start( )** method is called after init( ). It is also called to restart an applet after it has been stopped. Whereas init( ) is called once—the first time an applet is loaded—start( ) is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at start( ).

The **paint( )** method is called each time your applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. The paint( ) method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When stop( ) is called, the applet is probably running. You should use stop( ) to suspend threads that don't need to run when the applet is not visible. You can restart them when start( ) is called if the user returns to the page.

The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The stop( ) method is always called before destroy( ).

To output a string to an applet, we use drawString( ), which is a member of the **Graphics** class. Typically, it is called from within either update( ) or paint( ). It has the following general form:

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

To set the background color of an applet's window, use setBackground( ). To set the foreground color (the color in which text is shown, for example), use setForeground( ). These methods are defined by **Component**, and they have the following general forms:

void setBackground(Color newColor)

void setForeground(Color newColor)

Here, newColor specifies the new color. The class Color defines the constants shown here that can be used to specify colors:

Color.black, Color.magenta, Color.blue, Color.orange, Color.cyan, Color.pink, Color.darkGray, Color.red, Color.gray, Color.white, Color.green, Color.yellow, Color.lightGray

For Example:

setBackground(Color.green);

setForeground(Color.red);

EVENT HANDLING:

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

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.

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.

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.

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<u>Type</u>Listener(TypeListener el)

Here, ***<u>Type</u>*** 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 ***add<u>Key</u>Listener( )***.

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

public void removeTypeListener(TypeListener el)

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. It's one constructor is shown here:

EventObject(Object src)

Here, src is the object that generates this event. 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. Its getID( ) method can be used to determine the type of the event. The signature of this method is shown here:

int getID( )

The following table shows the most important of these event classes and provides a brief description of when they are generated.

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.

jkdirectory jkmaterials

Event Class	Description
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or losses 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.

SOURCE OF EVENTS:

The following table lists some of the user interface components that can generate the 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:

As we know, 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. The following table lists commonly used listener interfaces and provides a brief description of the methods that they define.

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved. (Added by Java 2, version 1.4)
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus. (Added by Java 2, version 1.4)
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

AWT PROGRAMMING:

The AWT contains numerous classes and methods that allow you 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. The following table lists some of the many AWT classes.

Class	Description
AWTEvent	Encapsulates AWT events.
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.
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.
FileDialog	Creates a window from which a file can be selected.
Frame	Creates a standard window that has a title bar, resize corners, and a menu bar.
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.
Menu	Creates a pull-down menu.
MenuBar	Creates a menu bar.

jkdirectory jkmaterials

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.

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.

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 superclass 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 you don't see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border.

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, you won't create Window objects directly. Instead, you 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 you 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.

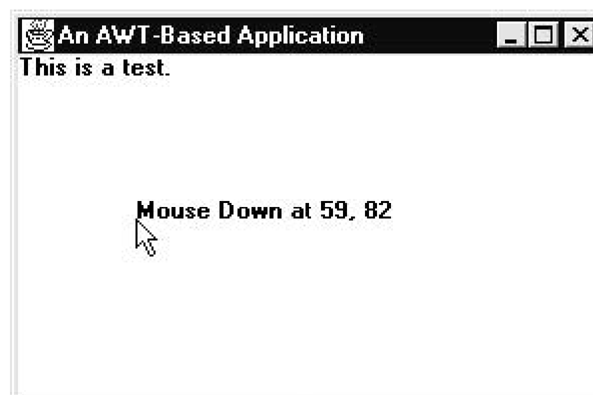
Example: Let us create a windowed program like this:

```
// Create an AWT-based application.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
// Create a frame window.
public class AppWindow extends Frame {
String keymsg = "This is a test.";
String mousemsg = "";
int mouseX=30, mouseY=30;
public AppWindow() {
addKeyListener(new MyKeyAdapter(this));
addMouseListener(new MyMouseAdapter(this));
addWindowListener(new MyWindowAdapter());
}
public void paint(Graphics g) {
g.drawString(keymsg, 10, 40);
g.drawString(mousemsg, mouseX, mouseY);
}
// Create the window.
public static void main(String args[]) {
AppWindow appwin = new AppWindow();
appwin.setSize(new Dimension(300, 200));
appwin.setTitle("An AWT-Based Application");
appwin.setVisible(true);
```

jkdirectory jkmaterials

```
}
}
class MyKeyAdapter extends KeyAdapter {
AppWindow appWindow;
public MyKeyAdapter(AppWindow appWindow) {
this.appWindow = appWindow;
}
public void keyTyped(KeyEvent ke) {
appWindow.keymsg += ke.getKeyChar();
appWindow.repaint();
};
}
class MyMouseAdapter extends MouseAdapter {
AppWindow appWindow;
public MyMouseAdapter(AppWindow appWindow) {
this.appWindow = appWindow;
}
public void mousePressed(MouseEvent me) {
appWindow.mouseX = me.getX();
appWindow.mouseY = me.getY();
appWindow.mousemsg = "Mouse Down at " + appWindow.mouseX +", " +
appWindow.mouseY;
appWindow.repaint();
}
}
class MyWindowAdapter extends WindowAdapter {
public void windowClosing(WindowEvent we) {
System.exit(0);
}
}
```

Sample output of the above program is as shown below:



Once created, a frame window takes on a life of its own. Notice that main( ) ends with the call to appwin.setVisible(true). However, the program keeps running until you close the window. In essence, when creating a windowed application, you will use main( ) to launch its top-level window. After that, your program will function as a GUI-based application, not like the console-based programs.

INTRODUCTION TO SWINGS:

Swing is a set of classes that 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 following are the Swing component classes that are used in common:

Class	Description
AbstractButton	Abstract superclass 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 (an 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.

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.

Handling Swing Controls Like:

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 in the table as follows:

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.

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)

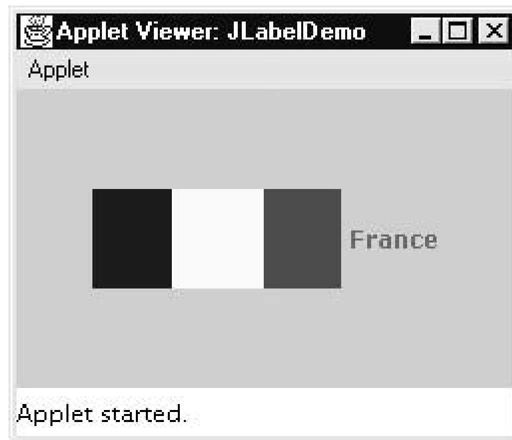
Here, i and s are the icon and text, respectively.

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 france.gif. 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() {
// Get content pane
Container contentPane = getContentPane();
// Create an icon
ImageIcon ii = new ImageIcon("france.gif");
// Create a label
JLabel jl = new JLabel("France", ii, JLabel.CENTER);
// Add label to the content pane
contentPane.add(jl);
}
}
```

Output from this applet is shown as follows:

jkdirectory jkmaterials



BUTTONS:

Swing buttons provide features that are not found in the Button class defined by the AWT. For example, you 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 you to control the behavior of buttons, check boxes, and radio buttons. For example, you can define different icons that are displayed for the component when it is disabled, pressed, or selected. Another icon can be used as a 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 unregister 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.

THE JBUTTON CLASS:

The JButton class provides the functionality of a push button. JButton allows an icon, a 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)

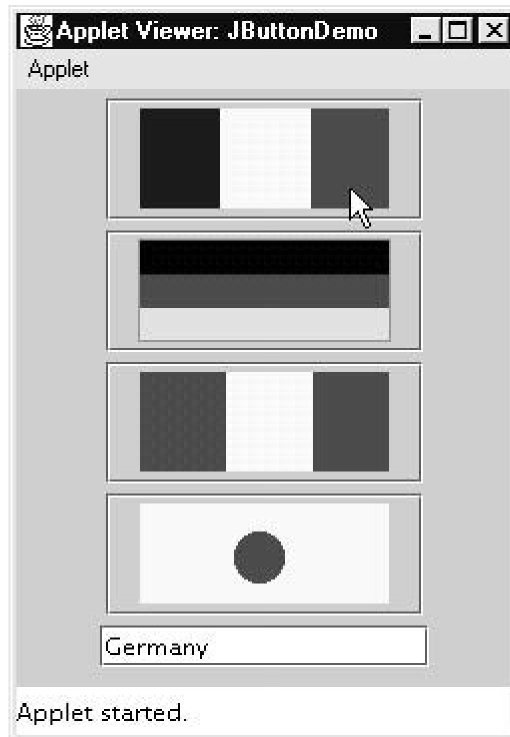
Here, s and i are the string and icon used for the button. The following example displays four push buttons and a text field. Each button displays an icon that represents the flag of a country. When a button is pressed, the name of that country is displayed in the text field. The applet begins by getting its content pane and setting the layout manager of that pane.

jkdirectory jkmaterials

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() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Add buttons to content pane
ImageIcon france = new ImageIcon("france.gif");
JButton jb = new JButton(france);
jb.setActionCommand("France");
jb.addActionListener(this);
contentPane.add(jb);
ImageIcon germany = new ImageIcon("germany.gif");
jb = new JButton(germany);
jb.setActionCommand("Germany");
jb.addActionListener(this);
contentPane.add(jb);
ImageIcon italy = new ImageIcon("italy.gif");
jb = new JButton(italy);
jb.setActionCommand("Italy");
jb.addActionListener(this);
contentPane.add(jb);
ImageIcon japan = new ImageIcon("japan.gif");
jb = new JButton(japan);
jb.setActionCommand("Japan");
jb.addActionListener(this);
contentPane.add(jb);
// Add text field to content pane
jtf = new JTextField(15);
contentPane.add(jtf);
}
public void actionPerformed(ActionEvent ae) {
jtf.setText(ae.getActionCommand());
}
}
```

Output for this applet is shown here:



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 you 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() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Add text field to content pane
jtf = new JTextField(15);
contentPane.add(jtf);
}
}
```

Output for this applet is shown here:

CHECK BOXES:

The JCheckBox class, which provides the functionality of a check box, is a concrete implementation of AbstractButton. Its immediate superclass 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.

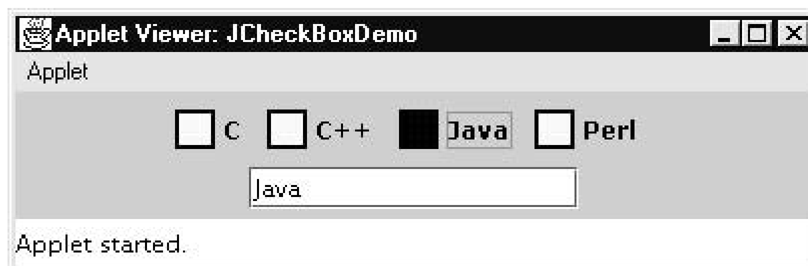
The following example illustrates how to create an applet that displays four check boxes and a text field. When a check box is pressed, its text is displayed in the text field. The content pane for the JApplet object is obtained, and a flow layout is assigned as its layout manager. Next, four check boxes 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 text 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() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
```

```
// Create icons
ImageIcon normal = new ImageIcon("normal.gif");
ImageIcon rollover = new ImageIcon("rollover.gif");
ImageIcon selected = new ImageIcon("selected.gif");
// Add check boxes to the content pane
JCheckBox cb = new JCheckBox("C", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("C++", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("Java", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("Perl", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
// Add text field to the content pane
jtf = new JTextField(15);
contentPane.add(jtf);
}
public void itemStateChanged(ItemEvent ie) {
JCheckBox cb = (JCheckBox)ie.getItem();
jtf.setText(cb.getText());
}
}
```

Output from this applet is shown here:



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. You can also type your selection into the text field. Two of JComboBox constructors are shown here:

JComboBox( )

JComboBox(Vector v)

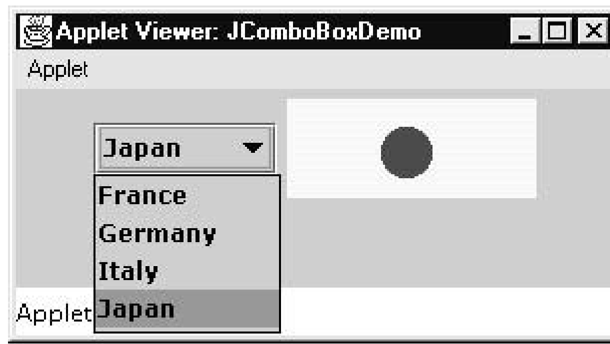
Here, v is a vector that initializes the combo box. Items are added to the list of choices via the addItem( ) method, whose signature is shown here:

void addItem(Object obj)

Here, obj is the object to be added to the combo box. The following example contains a combo box and a label. The label displays an icon. The combo box contains entries for “France”, “Germany”, “Italy”, and “Japan”. When a country is selected, the label is updated to display the flag for that country.

```
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 france, germany, italy, japan;
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Create a combo box and add it
// to the panel
JComboBox jc = new JComboBox();
jc.addItem("France");
jc.addItem("Germany");
jc.addItem("Italy");
jc.addItem("Japan");
jc.addItemListener(this);
contentPane.add(jc);
// Create label
jl = new JLabel(new ImageIcon("france.gif"));
contentPane.add(jl);
}
public void itemStateChanged(ItemEvent ie) {
String s = (String)ie.getItem();
jl.setIcon(new ImageIcon(s + ".gif"));
}
}
```

Output from this applet is show as follows:



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 content becomes 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. We will use its default constructor. 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 “Cities” and contains four buttons. Each button displays the name of a city. 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("Cities", new CitiesPanel());
jtp.addTab("Colors", new ColorsPanel());
jtp.addTab("Flavors", new FlavorsPanel());
getContentPane().add(jtp);
}
}
class CitiesPanel extends JPanel {
public CitiesPanel() {
JButton b1 = new JButton("New York");
add(b1);
```

```
JButton b2 = new JButton("London");
add(b2);
JButton b3 = new JButton("Hong Kong");
add(b3);
JButton b4 = new JButton("Tokyo");
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);
}
}
```

Output from this applet is shown in the following three illustrations:

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(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. You 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() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new BorderLayout());
// Add 400 buttons to a panel
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;
}
}
// Add panel to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(jp, v, h);
// Add scroll pane to the content pane
```

contentPane.add(jsp, BorderLayout.CENTER);

}

}

Output from this applet is shown here:

TREES:

A tree is a component that presents a hierarchical view of data. A user has the ability to expand or collapse individual subtrees in this display. 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 subtree is hidden, and the second method is called when a subtree becomes visible. Here are the steps that you 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.

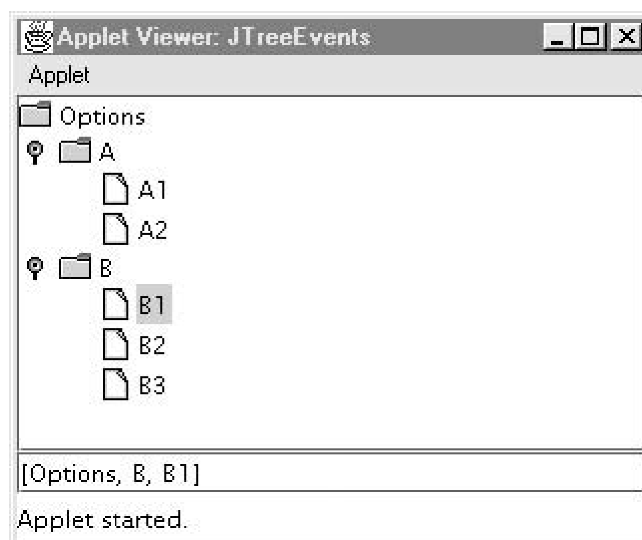
The tree is then provided as the argument to the JScrollPane constructor. This scroll pane is then added to the applet. Next, a text field is created and added to the applet. Information about mouse click events is presented in this text field. To receive mouse events from the tree, the addMouseListener( ) 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() {
// Get content pane
Container contentPane = getContentPane();
// Set layout manager
contentPane.setLayout(new BorderLayout());
// Create top node of tree
DefaultMutableTreeNode top = new DefaultMutableTreeNode("Options");
// Create subtree of "A"
DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
top.add(a);
DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");
a.add(a1);
DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");
a.add(a2);
// Create subtree of "B"
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);
// Create tree
tree = new JTree(top);
// Add tree to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(tree, v, h);
// Add scroll pane to the content pane
contentPane.add(jsp, BorderLayout.CENTER);
// Add text field to applet
jtf = new JTextField("", 20);
contentPane.add(jtf, BorderLayout.SOUTH);
// Anonymous inner class to handle mouse clicks
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("");
}
}
```

Output from this applet is shown here:



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. You can drag the cursor on column boundaries to resize columns. You 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[ ])

Here, data is a two-dimensional array of the information to be presented, and colHeads is a one-dimensional array with the column headings. Here are the steps for using a table in an applet:

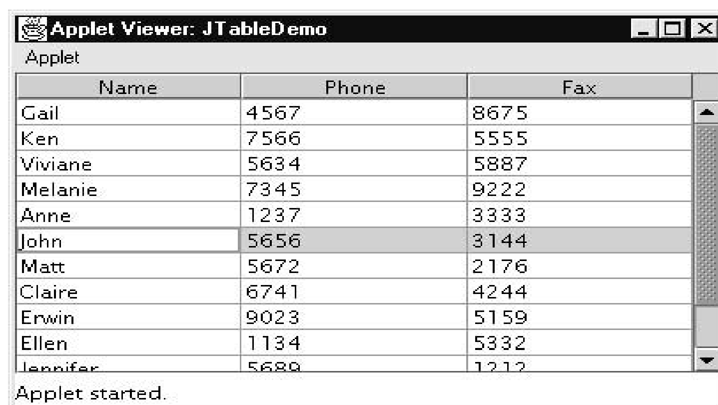
1. Create a JTable object.
2. Create a JScrollPane object. (The arguments to the constructor specify the table and the policies for vertical and horizontal scroll bars.)
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane of the applet.

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. You 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() {
// Get content pane
Container contentPane = getContentPane();
// Set layout manager
contentPane.setLayout(new BorderLayout());
// Initialize column headings
final String[] colHeads = { "Name", "Phone", "Fax" };
// Initialize data
final Object[][] data = {
{ "Gail", "4567", "8675" },
{ "Ken", "7566", "5555" },
{ "Viviane", "5634", "5887" },
{ "Melanie", "7345", "9222" },
{ "Anne", "1237", "3333" },
{ "John", "5656", "3144" },
{ "Matt", "5672", "2176" },
{ "Claire", "6741", "4244" },
{ "Erwin", "9023", "5159" },
{ "Ellen", "1134", "5332" },
{ "Jennifer", "5689", "1212" },
{ "Ed", "9030", "1313" },
{ "Helen", "6751", "1415" }
};
// Create the table
JTable table = new JTable(data, colHeads);
// Add table to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(table, v, h);
// Add scroll pane to the content pane
contentPane.add(jsp, BorderLayout.CENTER);
}
}
```

Output from this applet is shown here:



Name	Phone	Fax
Gail	4567	8675
Ken	7566	5555
Viviane	5634	5887
Melanie	7345	9222
Anne	1237	3333
John	5656	3144
Matt	5672	2176
Claire	6741	4244
Erwin	9023	5159
Ellen	1134	5332

jkdirectory jkmaterials

DIFFERENCE BETWEEN SWING AND AWT CONTROLS:

AWT:

1. AWT components are called Heavyweight component.
2. AWT components are platform dependent.
3. AWT does not support different look and feel.
4. AWT does not support advanced features like JTable, Jtabbed pane.
5. AWT uses peer components.
6. With AWT "peer" is a widget provided by the operating system, such as a button object or an entry field object.
7. AWT is a thin layer of code on top of the OS.
8. Using AWT, you have to implement a lot of things yourself.
9. AWT is more limited in supplying the GUI functionality on all platforms because not all platforms implement the same-looking controls in the same ways.
10. AWT is Java's original set of classes for building GUIs.
11. AWT is not truly portable.
12. It looks different and lays out inconsistently on different operating systems.

Advantages:

1. Use of native peers speeds component performance.
2. Most Web browsers support AWT classes so AWT applets can run without the Java plug-in (Applet Portability).
3. AWT components more closely reflect the look and feel of the OS they run on.

Disadvantages:

1. Use of native peers creates platform specific limitations. Some components may not function at all on some platforms.
2. AWT components do not support features like icons and tool-tips.

jkdirectory jkmaterials

Swing:

1. Swings are designed to solve AWT's problems.
2. Swings are called lightweight component because swing components sits on the top of AWT components and do the work.
3. Swings components are made in purely java and they are platform independent.
4. We can have different look and feel in Swing.
5. Swing has many advanced features like JTable, Jtabbed pane
6. Swing components are called "lightweight" because they do not require a native OS object to implement their functionality.
7. JDialog and JFrame are heavyweight, because they do have a peer. So components like JButton, JTextArea, etc., are lightweight because they do not have an OS peer.
8. With Swing, you would have only one peer, the operating system's window object. All of the buttons, entry fields, etc. are drawn by the Swing package on the drawing surface provided by the window object. This is the reason that Swing has more code.
9. Unlike AWT Swings has built in things.
10. Swing implements GUI functionality itself rather than relying on the host OS, it can offer a richer environment on all platforms Java runs on.
11. Drawing of components is done in java. It uses the AWT's components like window, frame, and dialog.
12. It lays out consistently on all operating systems and also uses AWT event handling.

Advantages:

1. Pure Java design provides for fewer platforms specific limitations (portability).
2. Pure Java design allows for a greater range of behavior for Swing components since they are not limited by the native peers that AWT uses.
3. Swing supports a wider range of features like icons and pop-up tool-tips for components.

Disadvantages:

1. Most Web browsers do not include the Swing classes, so the Java plug-in must be used (applet portability).
2. Swing components are generally slower and buggier than AWT, due to both the fact that they are pure Java and to video issues on various platforms. Since Swing components handle their own painting rather than using native API's like DirectX on Windows, you may run into graphical glitches (performance).
3. Even when Swing components are set to use the look and feel of the OS they are run on, they may not look like their native counterparts.

DEVELOPING HOME PAGE USING APPLET AND SWING:

To write a java swing applet you must import both the java.awt.* class library and also the javax.swing.* library. It is also necessary to import the javax.swing.JApplet library. If you expect to track any event such as a mouse click or the press of a button you will also need to import the java.awt.event.* library.

```
import java.awt.*; //import normal applet classes
import java.awt.event.*; //import event listeners
import javax.swing.*; //import swing components
import javax.swing.JApplet; //import swing applet interface
public class WhatEver extends JApplet
implements ActionListener {
public void init() { //initiallize the applet
getContentPane().add(label1, BorderLayout.NORTH);
pane1.setLayout(new GridLayout(1,2));
pane1.add(but1);
pane1.add(but2);
getContentPane().add(pane1, BorderLayout.SOUTH);
but1.addActionListener(this);
but2.addActionListener(this);
}
public void actionPerformed(ActionEvent event) {
Object source = event.getSource();
if(source == but1)
label1.setVisible(true);
else //if source == but2
label1.setVisible(false);
}
private JLabel label1 = new JLabel("Hello Java Swing World");
private JPanel pane1 = new JPanel();
private JButton but1 = new JButton("ON");
```

```
private JButton but2 = new JButton ("OFF");
}
```

You can create the applet with the Notepad editor found in the accessory section of the Windows Program menu. Make sure that you select "all files" before saving it as WhatEver.java. To compile this applet and execute it go to the command prompt and log to the area or disk where you have stored this simple applet.

Assume javac (java compiler) is located in the c:\j2sdk1.4.0_02\bin file folder and assume that java file is stored on a floppy disk in drive a: so my command line looked like the following when I compiled the program.

```
A:\>c:\j2sdk1.4.0_02\bin\javac WhatEver.java
```

This action creates the java class file for this applet called WhatEver.class. The java class is what executes from the webpage html file.

To execute this applet let us create a very simple html file let it is called as test.html and also stored on floppy disk in drive a:

```
<APPLET CODE=WhatEver.class WIDTH=200 HEIGHT=100>
</APPLET>
```

If you execute the test.html file you will see the applet in action. This simple applet merely states "Hello Java Swing World".

jkdirectory jkmaterials