UNIT-V

INTRODUCTION OF SWING

The Swing-related classes are contained in **javax.swing** and its subpackages, such as **javax.swing.tree**. Many other Swing-related classes and interfaces exist that are not examined in this chapter.
The remainder of this chapter examines various Swing components and illustrates them through sample applets.

JApplet

Fundamental to Swing is the **JApplet** class, which extends **Applet**. Applets that useSwing must be subclasses of **JApplet**. **JApplet** is rich with functionality that is notfound in **Applet**. For example, **JApplet** supports various "panes," such as the content pane, the glass pane, and the root pane. For the examples in this chapter, we will not be using most of **JApplet**'s enhanced features. However, one difference between **Applet** and **JApplet** is important to this discussion, because it is used by the sample applets in this chapter. 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.

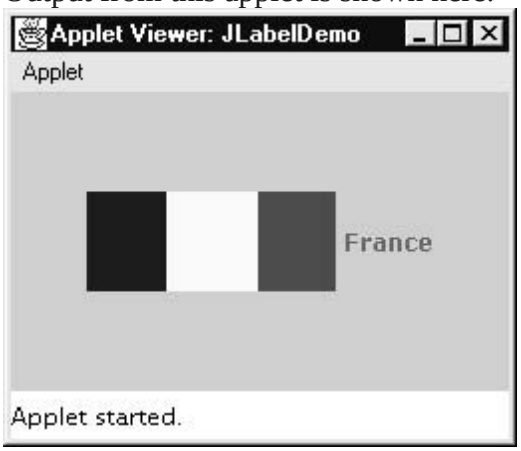
The icon and text associated with the label can be read and written by the following methods:
Icon getIcon( )
String getText( )
void setIcon(Icon *i*)
void setText(String *s*)
Here, *i* and *s* are the icon and text, respectively.
The following example illustrates how to create and display a label containing both an icon and a string. The applet begins by getting its content pane. Next, an **ImageIcon** object is created for the file **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 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:

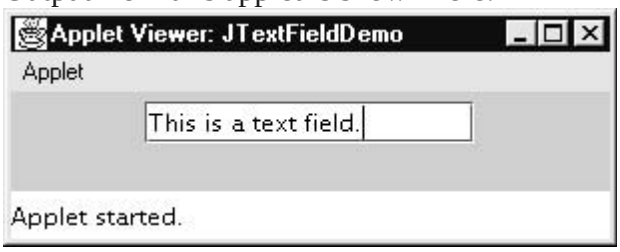
JTextField( )
JTextField(int *cols*)
JTextField(String *s*, int *cols*)
JTextField(String *s*)

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 from this applet is shown here:



Buttons

Swing buttons provide features that are not found in the **Button** class defined by theAWT. 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. Each is examined next.

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.

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.

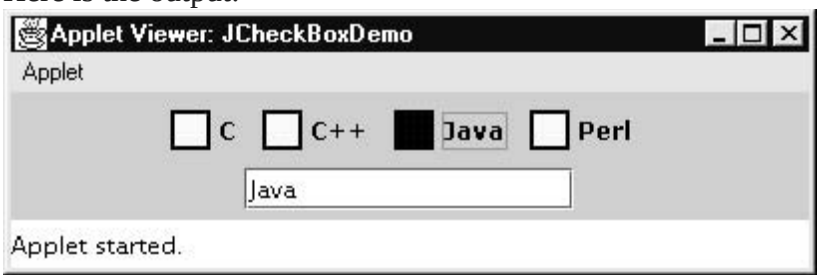
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo" width=400 height=50>
</applet>
*/
public class JCheckBoxDemo extends JApplet
implements ItemListener {
JTextField jtf;
public void init() {



// 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());
}
}
```

Here is the output:



Radio Buttons

Radio buttons are supported by the **JRadioButton** class, which 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:

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* is **true**, the button is initially selected. Otherwise, it is not.

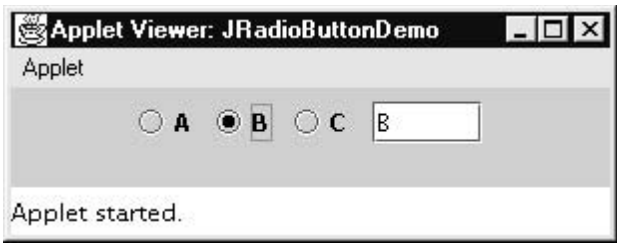
Radio buttons must be configured into a group. Only one of the buttons in that group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. The **ButtonGroup** class is instantiated to create 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.

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() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Add radio buttons to content pane
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);
// Define a button group
ButtonGroup bg = new ButtonGroup();
bg.add(b1);
bg.add(b2);
bg.add(b3);
// Create a text field and add it
// to the content pane
tf = new JTextField(5);
contentPane.add(tf);
}
public void actionPerformed(ActionEvent ae) {
tf.setText(ae.getActionCommand());
}
}
```

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**'s constructors are shown here:
JComboBox( )
JComboBox(Vector *v*)
Here, *v* is a vector that initializes the combo box.

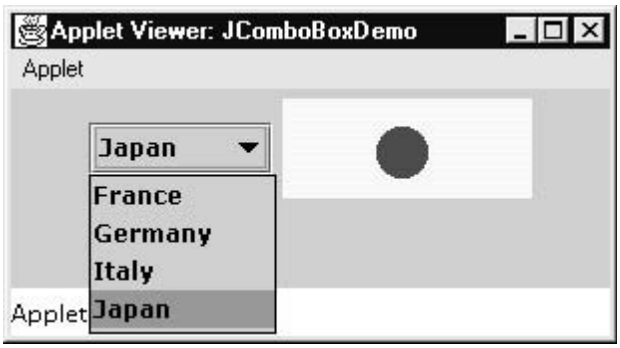
Items are added to the list of choices via the **addItem( )** method, whose signature is shown here:
void addItem(Object *obj*)
Here, *obj* is the object to be added to the combo box.

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 shown here:



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**. 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.

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

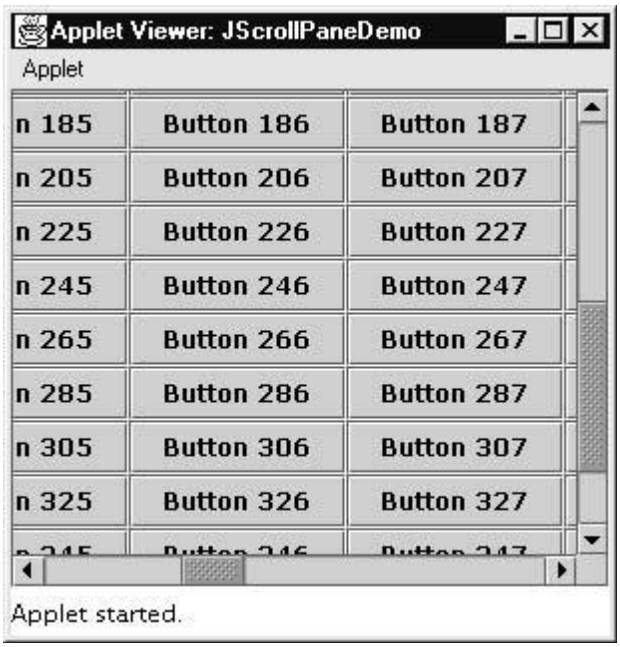
Here are the steps that you should follow to use a scroll pane in an applet:
1. Create a **JComponent** object.
2. Create a **JScrollPane** object. (The arguments to the constructor specify the component and the policies for vertical and horizontal scroll bars.)
3. Add the scroll pane to the content 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.

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.

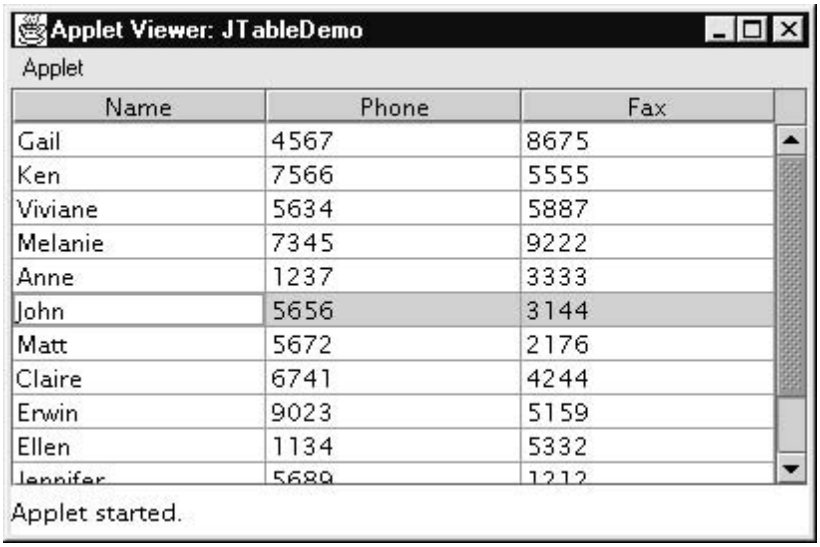
The following example illustrates how to create and use a table. The content pane of the **JApplet** object is obtained and a border layout is assigned as its layout manager. A one-dimensional array of strings is created for the column headings. This table has three columns. A two-dimensional array of 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);
}
}
```

Here is the output:



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

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, a software or hardware failure occurs, or an operation is completed. You 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(*Type*Listener *el*)
Here, *Type* is the name of the event and *el* is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener( )**. The method that registers a mouse motion listener is called **addMouseMotionListener( )**. When an event occurs, all registered listeners are notified and receive a copy of the event object. 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 add*Type*Listener(*Type*Listener *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 unregister an interest
in a specific type of event.

The general form of such a method is this:

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

Here, *Type* is the name of the event and *el* is a reference to the event listener. For example, to remove a keyboard listener, you would call **removeKeyListener( )**.
The methods that add or remove listeners are provided by the source that generates events. For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.

Event Classes

The classes that represent events are at the core of Java's event handling mechanism. Thus, we begin our study of event handling with a tour of the event classes. As you will see, they provide a consistent, easy-to-use means of encapsulating events.
At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**. It is the superclass for all events. Its 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( )

Additional details about **AWTEvent** are provided at the end of Chapter 22. At this point, it is important to know only that all of the other classes discussed in this section are subclasses of **AWTEvent**.

To summarize:

■ **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. Table 20-1 enumerates the most important of these event classes and provides a brief description of when they are generated. The most commonly

used constructors and methods in each class are described in the following sections.

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.

The 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. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

The MouseEvent Class

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 (Java 2, v1.4).

MouseEvent is a subclass of **InputEvent**. Here is one of its constructors.
MouseEvent(Component *src*, int *type*, long *when*, int *modifiers*,

int *x*, int *y*, int *clicks*, boolean *triggersPopup*)

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in *x* and *y*. The click count is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform. Java 2, version 1.4 adds a second constructor which also allows the button that caused the event to be specified.

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

int getX( )
int getY( )

Alternatively, you 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

Handling Keyboard Events

To handle keyboard events, you use the same general architecture as that shown in the mouse event example in the preceding section. The difference, of course, is that you will be implementing the **KeyListener** interface.

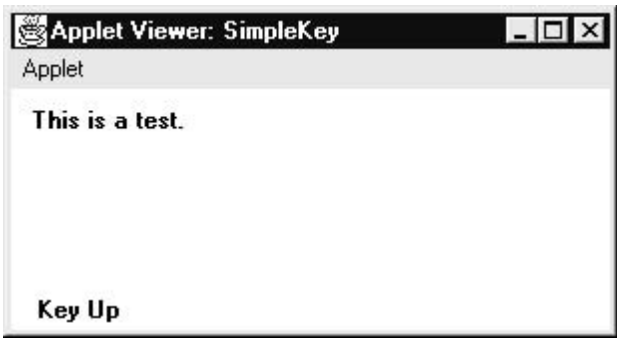
Before looking at an example, it is useful to review how key events are generated. When a key is pressed, a **KEY_PRESSED** event is generated. This results in a call to the **keyPressed( )** event handler. When the key is released, a **KEY_RELEASED** event is generated and the **keyReleased( )** handler 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 you care about are actual characters, then you can ignore the information passed by the key press and release events. However, if your 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 your 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 you don't, then your program will not receive any keyboard events.

```
// Demonstrate the key event handlers.
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);
}
}
```

Sample output is shown here:
If you want to handle the special keys, such as the arrow or function keys, you need to respond to them within the **keyPressed( )** handler. They are not available through **keyTyped( )**. To identify the keys, you use their virtual key codes. For example, the next applet outputs the name of a few of the special keys:



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 you want to receive and process only some of the events that are handled by a particular event listener interface. You 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 you 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. Table 20-4 lists 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

Demonstrate an adapter.

```
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");
}
}
```

Inner Classes

To understand 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.");
}
}
```

Anonymous Inner Classes

An *anonymous* inner class is one that is not assigned a name. This section illustrates how an anonymous inner class can facilitate the writing of event handlers. 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.

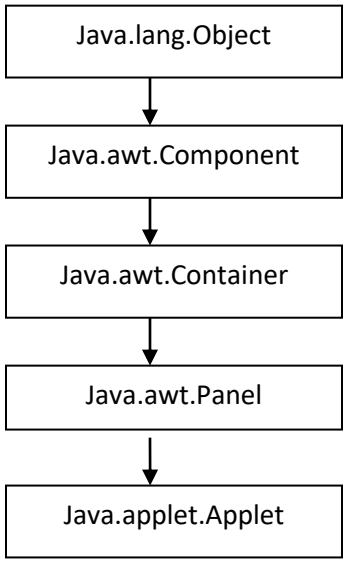
Because this anonymous inner class is defined within the scope of **AnonymousInnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, it can call the **showStatus( )** method directly. As just illustrated, both named and anonymous inner classes solve some annoying problems in a simple yet effective way. They also allow you to create more efficient code.

Applets

- Java programs are classified into two ways
  - Applications program
    - Application programs are those programs which are normally created, compiled and executed as similar to the other languages.
    - Each application program contains one main( ) method
    - Programs are created in a local computer.
    - Programs are compiled with javac compiler and executed in java interpreter.
  - Applet program
    - An applet is a special program that we can embedded in a web page such that the applet gains control over a certain part of the displayed page.
    - It is differ from application program
    - Applets are created from classes
    - An applet do not have main as an entry. Instead have several methods to control specific aspects of applet execution.

Class Hierarchy of Applet

- Every applet that we are creating must be a sub class of Applet.
- This Applet is extended from Panel.
- This Panel is extended from Container.
- The Container class extends Component class.
- Component class is extending from Object class which is parent of all Java API classes



About Java.awt.Component

- java.lang.Object
  - java.awt.Component
- public abstract class Component extends Object implements ImageObserver, MenuContainer, Serializable
- A *component* is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of components are the buttons, checkboxes, and scrollbars of a typical graphical user interface.

About Java.awt.Container

- java.lang.Object
  - java.awt.Component
    - **java.awt.Container**

- public class **Container** extends Component

- A generic Abstract Window Toolkit(AWT) container object is a component that can contain other AWT components.
- Components added to a container are tracked in a list. The order of the list will define the components' front-to-back stacking order within the container. If no index is specified

when adding a component to a container, it will be added to the end of the list (and hence to the bottom of the stacking order).

- Example : Add()

About Java.awt.Panel

- java.lang.Object
  - java.awt.Component
    - java.awt.Container
      - **java.awt.Panel**
- public class **Panel** extends Container implements Accessible
- Panel is the simplest container class. A panel provides space in which an application can attach any other component, including other panels.
- The default layout manager for a panel is the FlowLayout layout manager.

Java.applet.Applet

- java.lang.Object
  - java.awt.Component
    - java.awt.Container
      - java.awt.Panel
        - **java.applet.Applet**
- public class **Applet** extends Panel
  - An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application.
  - The Applet class must be the superclass of any applet that is to be embedded in a Web page or viewed by the Java Applet Viewer. The Applet class provides a standard interface between applets and their environment.

Example - 1

```
import java.awt.*;
import java.applet.*;
public class myapplet extends Applet
{
public void paint(Graphics g)
{
g.drawString("Welcome to applet",30,30);
```

```
}
}
```

Html file

```
<html>
<head>
<body>
<applet code = "myapplet.class" width=200 height=100>
abcd
</applet>

</body>
</head>
```

Execution procedure of applet program

- Save the above program with myapplet.java
- Compile the program using
    - Javac myapplet.java
- In order to run the program there are two methods
    - Using web browser
        - executing the applet through html within a java compatible browser such as HotJava, Netscape Navigator or Internet explorer. to execute this method, we need write the HTML file with Applet tag.
        - Save the program with test.html.
        - Open web browser, and type path of file at address bar
    - From console
        - At the console window give the following command
            - Appletviewer test.html

Method – 2

```
import java.awt.*;
import java.applet.*;
public class myapplet extends Applet
{
public void paint(Graphics g)
{
g.drawString("Welcome to applet",30,30);
}
}

/*
<applet code = "myapplet.class" width = 200 height = 100>
```

</applet>
*/

- Save the above program with myapplet.java
- Compile it using javac myapplet.java
- Run using appletviewer myapplet.java

Architecture of an Applet

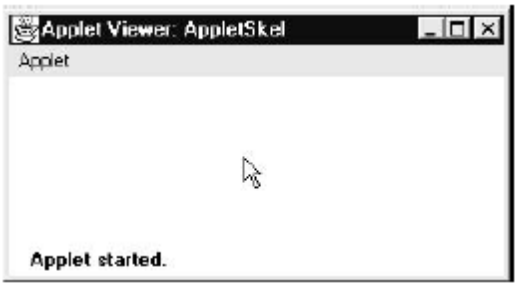
- Since applet is an window based program. Its architecture is different from the console based programs.
- In window based program we must understand few concepts:
  - First, applets are event driven.
  - how the event-driven architecture impacts the design of an applet.
    - An applet resembles a set of interrupt service routines.
  - Here is how the process works.
    - An applet waits until an event occurs. The AWT notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return control to the AWT. This is a crucial point. For the most part, your applet should not enter a "mode" of operation in which it maintains control for an extended period. Instead, it must perform specific actions in response to events and then return control to the AWT run-time system. In those situations in which your applet needs to perform a repetitive task on its own (for example, displaying a scrolling message across its window), you must start an additional thread of execution.
  - Second, the user initiates interaction with an applet—not the other way around.
    - As you know, in a no windowed program, when the program needs input, it will prompt the user and then call some input method, such as **readLine( )**. This is not the way it works in an applet. Instead, the user interacts with the applet as he or she wants, when he or she wants. These interactions are sent to the applet as events to which the applet must respond.
    - For example, when the user clicks a mouse inside the applet's window, a mouse-clicked event is generated. If the user presses a key while the applet's window has input focus, a keypress event is generated. As you will see in later chapters, applets can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated.

Applet Skelton

- 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. efault 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
}
/* 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

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( )**

init( )

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.

start( )

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( )**.

paint( )

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. Or the applet window may be minimized and then restored. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called. The **paint( )** method has one parameter of type **Graphics**.

This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

stop( )

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.

destroy( )

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( )**.

Overriding update( )

In some situations, your applet may need to override another method defined by theAWT, called **update( )**. This method is called when your applet has requested that a

portion of its window be redrawn. The default version of **update( )** first fills an applet with the default background color and then calls **paint( )**. If you fill the background using a different color in **paint( )**, the user will experience a flash of the default background each time **update( )** is called—that is, whenever the window is repainted.

One way to avoid this problem is to override the **update( )** method so that it performs all necessary display activities. Then have **paint( )** simply call **update( )**. Thus, for some applications, the applet skeleton will override **paint( )** and **update( )**, as shown here:

```
public void update(Graphics g) {
// redisplay your window, here.
}
public void paint(Graphics g) {
update(g);
}
```

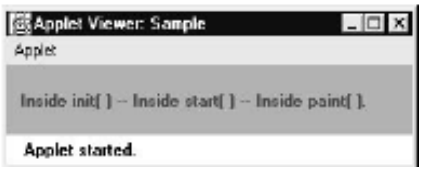
- To output a string to an applet, use **drawString( )**, which is a member of the **Graphics** class. Typically, it is called from within either **update( )** or **paint( )**. It has the following general form:
  - **void drawString(String *message*, int *x*, int *y*)**
    - Here, *message* is the string to be output beginning at *x,y*. In a Java window, the upper-left corner is location 0,0. The **drawString( )** method will not recognize newline characters. If you want to start a line of text on another line, you must do so manually, specifying the precise X,Y location where you want the line to begin.
- To set the background color of an applet's window, use **setBackground( )**.
- To set the foreground color 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.darkGra
- Color.red
- Color.gray
- Color.white
- Color.green
- Color.yellow
- Color.lightGray

- For example, this sets the background color to green and the text color to red:
    - **setBackground(Color.green);**
    - **setForeground(Color.red);**
  - We has to set the foreground and background colors is in the **init( )** method
  - we can change these colors during execution of program also
  - The default foreground color is black.
  - The default background color is light gray.
  - we can obtain the current settings for the background and foreground colors by calling **getBackground( )** and **getForeground( )**, respectively. They are also defined by **Component** and are shown here:
    - **Color getBackground( )**
    - **Color getForeground( )**
  -

Program

```
/* A simple applet that sets the foreground and
background colors and outputs a string. */
import java.awt.*;
import java.applet.*;
/*
<applet code="Sample" width=300 height=50>
</applet>
*/
public class Sample extends Applet{
String msg;
// set the foreground and background colors.
public void init() {
setBackground(Color.cyan);
setForeground(Color.red);
msg = "Inside init( ) --";
}
// Initialize the string to be displayed.
public void start() {
msg += " Inside start( ) --";
}
// Display msg in applet window.
public void paint(Graphics g) {
msg += " Inside paint( ).";
g.drawString(msg, 10, 30);
}
}
```



- **Using Repaint method**
  - As a general rule, an applet writes to its window only when its **update( )** or **paint( )** method is called by the AWT.

- How can the applet itself cause its window to be updated when its information changes? For example, if an applet is displaying a moving banner, what mechanism does the applet use to update the window each time this banner scrolls?
- It cannot create a loop inside **paint( )** that repeatedly scrolls the banner
- The **repaint( )** method is defined by the AWT. It causes the AWT run-time system to execute a call to your applet's **update( )** method, which, in its default implementation, calls **paint( )**
- The **repaint( )** method has four forms.
- The simplest version of **repaint( )** is shown here:
    - **void repaint( )**
- This version causes the entire window to be repainted.
- The following version specifies a region that will be repainted:
    - **void repaint(int *left*, int *top*, int *width*, int *height*)**
- Here, the coordinates of the upper-left corner of the region are specified by *left* and *top*, and the width and height of the region are passed in *width* and *height*. These dimensions are specified in pixels
- Calling **repaint( )** is essentially a request that your **applet** be repainted sometime soon. However, if your system is slow or busy, **update( )** might not be called immediately. Multiple requests for repainting that occur within a short time can be collapsed by the AWT in a manner such that **update( )** is only called sporadically. This can be a problem in many situations, including animation, in which a consistent update time is necessary. One solution to this problem is to use the following forms of **repaint( )**:
    - **void repaint(long *maxDelay*)**
    - **void repaint(long *maxDelay*, int *x*, int *y*, int *width*, int *height*)**
- Here, *maxDelay* specifies the maximum number of milliseconds that can elapse before **update( )** is called

Program

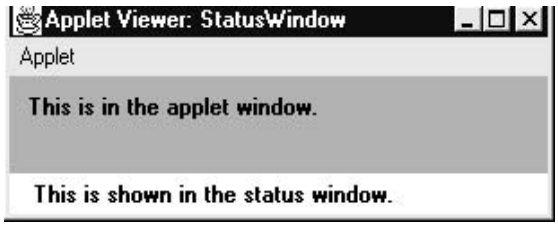
```
/* A simple banner applet.
This applet creates a thread that scrolls
the message contained in msg right to left
across the applet's window.
*/
import java.awt.*;
import java.applet.*;
/*
```

```
<applet code="SimpleBanner" width=300 height=50>
</applet>
*/
public class SimpleBanner extends Applet implements Runnable {
String msg = " A Simple Moving Banner.";
Thread t = null;
int state;
boolean stopFlag;
// Set colors and initialize thread.
public void init() {
setBackground(Color.cyan);
setForeground(Color.red);
}
// Start thread
public void start() {
t = new Thread(this);
stopFlag = false;
t.start();
}
// Entry point for the thread that runs the banner.
public void run() {
char ch;
// Display banner
for( ; ; ) {
try {
repaint();
Thread.sleep(250);
ch = msg.charAt(0);
msg = msg.substring(1, msg.length());
msg += ch;
if(stopFlag)
break;
} catch(InterruptedException e) {}
}
}
// Pause the banner.
public void stop() {
stopFlag = true;
t = null;
}
// Display the banner.
public void paint(Graphics g) {
g.drawString(msg, 50, 30);
}
}
```

Using Status window

- In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running.
- To do so, call **showStatus( )** with the string that you want displayed. The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors. The status window also makes an excellent debugging aid, because it gives you an easy way to output information about your applet.

```
// Using the Status Window.
import java.awt.*;
import java.applet.*;
/*
<applet code="StatusWindow" width=300 height=50>
</applet>
*/
public class StatusWindow extends Applet{
public void init() {
setBackground(Color.cyan);
}
// Display msg in applet window.
public void paint(Graphics g) {
g.drawString("This is in the applet window.", 10, 20);
showStatus("This is shown in the status window.");
}
}
```



Syntax of Applet tag in HTML

The syntax for the standard APPLET tag is shown here. Bracketed items
are optional.

```
< APPLET
[CODEBASE = codebaseURL]
CODE = appletFile
[ALT = alternateText]
[NAME = appletInstanceName]
WIDTH = pixels HEIGHT = pixels
[ALIGN = alignment]
[VSPACE = pixels] [HSPACE = pixels]
>
[< PARAM NAME = AttributeName VALUE = AttributeValue>]
[< PARAM NAME = AttributeName2 VALUE = AttributeValue>]
. . .
[HTML Displayed in the absence of Java]
```

</APPLET>

CODEBASE

CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read.

CODE

CODE is a required attribute that gives the name of the file containing your applet's compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.

ALT

The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser understands the APPLET tag but can't currently run Java applets. This is distinct from the alternate HTML you provide for browsers that don't support applets.

NAME

NAME is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use **getApplet( )**, which is defined by the **AppletContext** interface.

WIDTH AND HEIGHT

WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area. ALIGN ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

VSPACE AND HSPACE

These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.

PARAM NAME AND VALUE

The PARAM tag allows you to specify applet specific arguments in an HTML page. Applets access their attributes with the **getParameter( )** method.

HANDLING OLDER BROWSERS

Some very old web browsers can't execute applets and don't recognize the APPLET tag. Although these browsers are now nearly extinct (having been replaced by Java-compatible ones), you may need to deal with them occasionally. The best way to design your HTML page to deal with such browsers is to include HTML text and markup within your <applet></applet> tags.

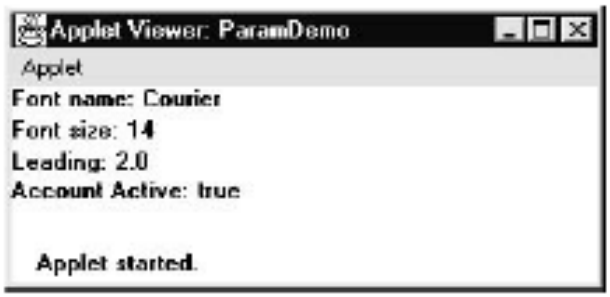
If the applet tags are not recognized by your browser, you will see the alternate markup. If Java is available, it will consume all of the markup between the <applet></applet> tags and disregard the alternate markup.
Here's the HTML to start an applet called **SampleApplet** in Java and to display a message in older browsers:

```
<applet code="SampleApplet" width=200 height=40>
If you were driving a Java powered browser,
you'd see &quote;A Sample Applet&quote; here.<p>
</applet>
```

Passing Parameters to Applets:

- the APPLET tag in HTML allows you to pass parameters to your applet. To retrieve a parameter, use the **getParameter( )** method.
- It returns the value of the specified parameter in the form of a **String** object. Thus, for numeric and **boolean** values, you will need to convert their string representations into their internal formats.
- // Use Parameters

```
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/
public class ParamDemo extends Applet{
String fontName;
int fontSize;
float leading;
boolean active;
// Initialize the string to be displayed.
public void start() {
String param;
fontName = getParameter("fontName");
if(fontName == null)
fontName = "Not Found";
param = getParameter("fontSize");
try {
if(param != null) // if not found
fontSize = Integer.parseInt(param);
else
fontSize = 0;
} catch(NumberFormatException e) {
```



```
fontSize = -1;
}
param = getParameter("leading");
try {
if(param != null) // if not found
leading = Float.valueOf(param).floatValue();
else
leading = 0;
} catch(NumberFormatException e) {
leading = -1;
}
param = getParameter("accountEnabled");
if(param != null)
active = Boolean.valueOf(param).booleanValue();
}
// Display parameters.
public void paint(Graphics g) {
g.drawString("Font name: " + fontName, 0, 10);
g.drawString("Font size: " + fontSize, 0, 26);
g.drawString("Leading: " + leading, 0, 42);
g.drawString("Account Active: " + active, 0, 58);
}
}
```