Unit IV
Applets and Swing

Applet Basics

- Applets are small programs that are designed for transmission over the Internet and run within a browser
- There are two general varieties
    1. AWT Based (It is original toolkit)
    2. Swing Based (Lightweight alternative)
- There are two ways to run an applet
    1. Using a browser
    2. Using the appletviewer tool provided by JDK

How Applets differ from Console-based Programs

- All applets created are subclasses of Applet
- Applets do not need a main method
- Applets must be run under an applet viewer or a Java-compatible browser
- User I/O is not accomplished with Java's stream I/O classes. Instead, Applets use interface provided by the AWT or Swing
- Applets are event-driven. That is, an Applet waits until an event occurs.
- Run-time system notifies the applet about an event by calling event handler
- User initiates interaction with the Applet. These interactions are sent to the applet as events to which the applet must respond. In case of console-based program, when the program needs input, it will prompt the user.

Example:

```
import java.awt.*;
import java.applet.*;
public class SampleApplet extends Applet {
public void paint(Graphics g) {
g.dawString("Java Applets are easy and simple", 20,20); } }
```

The first import statement imports Abstract Window Toolkit classes. Applets interact with the users through AWT. AWT contains support for a window-based graphical user interface. The second import statement imports the applet package. This package contains Applet class and we must create a sub-class of Applet class. this program can be complied in the same way as a console-based program however, it requires an HTML file to load the applet. The HTML code for the above example is

```
<applet code="SampleApplet.class" width=200 height=300>
</applet>
```

A Complete Applet Skeleton

All applets must override a set of methods that provide the basic mechanism to control the execution. These lifecycle methods are init(), start(), stop() and destroy() which are defined by the Applet class. The paint() method is inherited from AWT component class.

```
import java.awt.*;
import java.applet.*;
public class AppSk extends Applet
{
public void init() { // for initialization}
public void start(){ //to start or resume }
public void stop() { // to suspend execution }
```

```
public void destroy() { // to perform shutdown activities }
public void paint() { //to redisplay contents of window }
}
```

- init () is the first method to be called. It contains code to initialize variables and perform any other startup activities
- start() is called after init(). It is also called to re-start the applet after it has been stopped
- start() might be called more than once during the life cycle of an applet.
- When page containing an applet is left, the stop() method is called.
- However stop() dose not mean that applet is terminated. It might be restarted with a call to start() method if user returns to page
- The destroy() method is called when the applet is no longer needed.
- paint() method is called by the run-time system or when ever an applet must re-draw its output.
- The paint() method is called automatically whenever the window needs to be refreshed. The programmer never calls paint() .
- **repaint()**:-Programmer calls repaint() in order to obtain a rendering. repaint() then again call paint() to service repaint() method.
- The **update()** method : It is called e applet when some portion of the applet window is to be redrawn

Using the 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. This is done by calling showStatus() method defined by Applet. Its syntax is

```
showStatus(String msg);
```

Example

```
import java.awt.*;
import java.applet.*;
public class StatusWindow extends Applet {
public void paint(Graphics g) {
g.dawString("This is an Applet Window", 20,20);
showStatus("This is shown in the status window"); } }
```

Passing Parameters to Applets

Parameters can be passes to an applet using PARAM attribute of APPLET tag , specifying the name of the parameter and the value. Parameters are received by an applet using getParameter() method defend by Applet. Its general form is :

```
String getParameter(String paramName);
```

Here, paramName is the name of the parameter. It returns the value in the form of a String object. If any other types of parameters are passed, they need to be converted to String.

Example

```
import java.awt.*;
import java.applet.*;
/* <applet code =Param.class width =300 height=80>
<param name=author value="Herbert Shildt" >
<param name=purpose value="To Demonstrate Parameters" >
<param name=version  value=2 > </applet> */
```

```
public class Param extends Applet {
String author, purpose;
int ver;
public void start(){
String temp;
author=getParameter("author");
if (author==null)
author="not found"
purpose=getParameter("purpose");
if (purpose==null)
 purpose="not found";
try{
   if(temp!=null)
      ver=Integer.parseInt(temp);
   else
      ver=0; }
catch(NumberFormatException e)
{ ver=-1; } }
public void paint(Graphics g) {
g.dawString("Purpose : "+purpose, 10,20);
g.dawString("By : "+author, 10,40);
g.dawString("Version  : "+ver, 10,60); }}
```

The Applet Class

- All applets are the subclasses of the Applet class.
- Applet inherits Component, Container and Panel superclasses defined by the AWT.
- An applet has access to full functionality of the AWT

Method	Description
void destroy()	Called by the browser just before an applet is terminated. If overridden, this method will perform any cleanup before destruction.
AccessibleContext getAccessibleContext()	Returns the accessibility context for the invoking object.
AppletContext getAppletContext()	Returns the context associated with the applet.
String getAppletInfo()	Returns the string that describes the applet.
AudioClip getAudioClip(URL url)	Returns the AudioClip object that encapsulates the audio clip found at the location specified by url.
AudioClip getAudioClip(URL url, String ClipName)	Returns the 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.
Locate getLocate()	Returns a Locate object that is used by various locate sensitive classes and methods.

String getParameter(String paramName)	Returns the parameter associated with paramName. Null is returned if the specified parameter is not found.
String[][] getParameterInfo()	If overridden, returns a String table that describes the parameters recognised by the applet. Each entry in the table must consist of three strings that contain the name of the parameter, a description and an explanation of its purpose. The default implementation returns null.
void init()	Called when an applet begins execution. It is the first method called for an applet.
boolean isActive()	Returns true if the applet has been started. It returns false if the applet has been stopped.
Static final AudioClip newAudioclip(URL url)	Returns an AudioClip object that encapsulates the audio clip found at the location specified by url. This method is similar to getAudioClip() except that it is static and cab be executed without the need for an Applet object.
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 th name specified by clipName, the clip is played.
void resize(Dimension dim)	Resizes the applet according to the dimension specified by dim. Dimension is a class stored inside java.awt. It contains two integer fields: width and height.
final void setStub(AppletStub stubObj)	Makes stubObj the stub for the applet. This method is used by the run-time system and is not usually called by the applet. A stub is a small piece of code that provides linkage between applet and the browser.
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().

Event Handling

Java applets are event-driven. Most events to which the program will respond are generated by the user. These events are passed to the program in a variety of ways. There are several types of events, including those generated by the mouse, the keyboard, and various controls like push buttons. AWT-based events are supported by the **java.awt.event** package.

The Delegation Event Model

This model defines standard and consistent mechanisms to generate and process events. A **source** generates an event and sends it to one or more **listeners**. Listener 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.

Events

In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples.

Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, software or hardware failure occurs, or an operation is completed.

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 addTypeListener(TypeListener el)
```

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

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)
```

Here, Type is the name of the event, and el is a reference to the event listener. For example, to remove a keyboard listener, removeKeyListener( ) is to be called.

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. Thus, a discussion of event handling must begin with the event classes At the root of the Java event class hierarchy is EventObject, which is in java.util. It is the superclass for all events. The

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

Event Listener Interface

Event listeners receive event notifications. Listeners for AWT-based events 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 arguments. Commonly used event classes in java.awt.event are given below:

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Commonly used event listener interfaces are listed in the table below

Interface	Description
ActionListener	Defines one method to receive action events. Action events are generated by push buttons menus etc
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.
TextListener	Defines one method to recognize when a text value changes.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Applet Programming Using Delegation Event Model

Using the delegation event model is actually quite easy. Just follow these two steps:

1. Implement the appropriate interface in the listener so that it will receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

A source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.

Handling Mouse and Mouse Motion Events

To handle mouse events, we must implement the MouseListener and the MouseMotionListener interfaces. The MouseListener interface defines five methods.

void mouseClicked(MuseEvent me)	Invoked when a mouse button is clicked
void mouseEntered(MouseEvent me)	Invoked when mouse enters a component
void mouseExited(MouseEvent me)	Invoked when mouse exits from a component
void mousePressed(MouseEvent me)	Invoked when a mouse button is pressed
void mouseReleased(MouseEvent me)	Invoked when a mouse button is released

If a mouse button is clicked, mouseClicked() is invokeThe MouseListener interface defines five methods. If a mouse button is clicked, mouseClicked() is invoked.

Example

```
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</applet>
*/
public class MouseEvents extends Applet
implements MouseListener, MouseMotionListener {
String msg = "";
int mouseX = 0, mouseY = 0; // coordinates of mouse

public void init() {
addMouseListener(this);
addMouseMotionListener(this); }
public void mouseClicked(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse clicked.";
repaint(); }
```

```
public void mouseEntered(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse entered.";
repaint(); }

public void mouseExited(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse exited.";
repaint(); }

public void mousePressed(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Down";
repaint(); }

public void mouseReleased(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Up";
repaint(); }
public void paint(Graphics g) {
g.drawString(msg, mouseX, mouseY); } }
```

The transient and volatile Modifiers

These modifiers handle special situations. When an instance variable is declared as transient, then its value need not persist when an object is stored. A transient field does not affect the state of an object. The volatile modifier tells the compiler that a variable can be changed unexpectedly by other parts of the program. One of these situations involve multithreaded programs. In a multithreaded program, sometimes two or more threads share same variable. In such situations that variable can be declared as volatile.

instanceof : Sometimes it is useful to know the type of an object during run time. It is useful in the following situations:

1. One thread generates various types of objects and another thread may process these objects. In this situation, it is useful for the processing thread to know the type of each object received.
2. Knowledge of an object type at run time is important in case of casting. A superclass reference can refer to subclass objects, it is not always possible to know at compile time whether a cast is valid or not.

The instanceof keyword addresses these types of situations. Its general form is

objref instanceof type → here, objref is a reference to an instance of a class, and type is a class or interface. If objref is of specific type, then instanceof operator returns true else it returns false.

strictfp: When Java 2 was released, the floating point model was relaxed slightly. It doesn't require the truncation of certain intermediate values that occur during computation. This prevents overflow and underflow. By modifying a class, method or interface with strictfp, we can ensure floating point calculations take place precisely.

assert: The assert keyword is used to create an assertion, which is a condition that is expected to be true during the execution of the program. For example, we may require a method which always returns a positive integer value. We can test this by asserting that the return value is greater than zero using an assert statement. At run time, if the condition is actually true, no other action takes place. However, if the condition is false, an AssertionError is thrown. Assertion is often used in testing to verify that some expected condition is actually met.
Syntax: assert condition;
Exampe: assert n>0;

Native Methods
We may require to call a subroutine that is written in a language other than Java. Such subroutine will exist as executable code. This is called native code. We may wish to call a native method for faster execution or we may use a special third party library such as a statistical package. Java provides native keyword, which is used to declare native code methods. Once declared, these methods can be called from a Java program just as a Java method. To declare a native method, one just needs native keyword and method declaration but body of the method is not needed.
Example : `public native int meth();`

The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Choice lists
- Scroll bars
- Text editing

These controls are subclasses of **Component**.

Adding and Removing Controls
To include a control in a window, it must be added to the window. To do this, first an instance of the desired control must be created and then it must be added it to a window by calling **add( )**, which
is defined by **Container**.

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

Here, *compObj* is an instance of the control to be added. A reference to *compObj* is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.
A control which is no longer needed can be removed from the window. To do this, **remove( )** is used. This method is also defined by **Container**. It has this general form:

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

Here, *obj* is a reference to the control you want to remove. All controls can be removed by calling **removeAll( )**.

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

The HeadlessException

Most of the AWT controls described in this chapter now have constructors that can throw a **HeadlessException** when an attempt is made to instantiate a GUI component in a non-interactive environment (such as one in which no display, mouse, or keyboard is present). This exception can be used to write code that can adapt to non-interactive environments.

Labels

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

Example:

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

The text in a label can be changed by using the **setText( )** method. The caption of the current label can by calling **getText( )**. These methods are shown here:

void setText(String *str*)

String getText( )

For **setText( )**, *str* specifies the new label. For **getText( )**, the current label is returned. The alignment of the string within the label can be changed by calling **setAlignment( )**.

To obtain the current alignment, call **getAlignment( )**. The methods are as follows:

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

Here, *how* must be one of the alignment constants shown earlier.

The following example creates three labels and adds them to an applet window:

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

Using Buttons

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

Button( ) throws HeadlessException
Button(String *str*) throws HeadlessException
The first version creates an empty button. The second creates a button that contains *str* as a label.
After a button has been created, its label can be set by calling **setLabel( )**. The text associated with a button can be retrieved by calling **getLabel( )**.
These methods are as follows:
void setLabel(String *str*)
String getLabel( )
Here, *str* becomes the new label for the button.

Handling Buttons
Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component.
Each listener implements the **ActionListener** interface. That interface defines the **actionPerformed()** method, which is called when an event occurs. An **ActionEvent** object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the *action command string* associated with the button. By default, the action command string is the label of the button. Usually, either the button reference or the action command string can be used to identify the button.
Here is an example that creates three buttons labeled “Yes”, “No”, and “Undecided”. Each time one is pressed, a message is displayed that reports which button has been pressed.

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

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

Applying Check Boxes

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

Checkbox supports these constructors:

Checkbox( ) throws HeadlessException

Checkbox(String *str*) throws HeadlessException

Checkbox(String *str*, boolean *on*) throws HeadlessException

The first form creates a check box whose label is initially blank. The state of the check box is unchecked. The second form creates a check box whose label is specified by *str*. The state of the check box is unchecked. The third form allows to set the initial state of the check box. If *on* is **true**, the check box is initially checked; otherwise, it is cleared. The value of *on* determines the initial state of the check box.

To retrieve the current state of a check box, call **getState( )**. To set its state, call **setState( )**. The current label associated with a check box can be obtained by calling **getLabel( )**. To set the label, **setLabel( )** can be called. These methods are as follows:

boolean getState( )

void setState(boolean *on*)

String getLabel( )

void setLabel(String *str*)

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

Handling Check Boxes

Each time a check box is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener** interface. That interface defines the **itemStateChanged( )** method. An **ItemEvent** object is supplied as the argument to this method. It contains information about the event (for example, whether it was a selection or de-selection).

CheckboxGroup

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called *radio buttons*, because they act like the station selector on a car radio—only one station can be selected at any one time. To create a set of mutually exclusive check boxes, Check box groups are objects of type **CheckboxGroup**. Only the default constructor is defined, which creates an empty group. One can determine which check box in a group is currently selected by calling
getSelectedCheckbox( ). A check box can that is selected can be obtained by calling **setSelectedCheckbox( )**. These methods are as follows:
Checkbox getSelectedCheckbox( )i
void setSelectedCheckbox(Checkbox *which*)

Choice Controls
The **Choice** class is used to create a *pop-up list* of items from which the user may choose. Thus, a **Choice** control is a form of menu. When inactive, a **Choice** component takes up only enough space to show the currently selected item. When the user clicks on it, the whole list of choices pops up, and a new selection can be made. Each item in the list is a string that appears as a left-justified label in the order it is added to the **Choice** object. **Choice** only defines the default constructor, which creates an empty list. To add a selection to the list, call **add( )**. It has this general form:
void add(String *name*)
Here, *name* is the name of the item being added. Items are added to the list in the order in which calls to **add( )** occur.
To determine which item is currently selected, you may call either **getSelectedItem( )** or **getSelectedIndex( )**. These methods are shown here:
String getSelectedItem( )
int getSelectedIndex( )
The **getSelectedItem( )** method returns a string containing the name of the item.
getSelectedIndex( ) returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected.

To obtain the number of items in the list, call **getItemCount( )**. You can set the currently selected item using the **select( )** method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:
int getItemCount( )
void select(int *index*)
void select(String *name*)
Given an index, the name associated with the item at that index can be obtained by calling **getItem()**, which has this general form:
String getItem(int *index*)
Here, *index* specifies the index of the desired item.

Using Lists
The **List** class provides a compact, multiple-choice, scrolling selection list. Unlike the **Choice** object, which shows only the single selected item in the menu, a **List** object can be

constructed to show any number of choices in the visible window. It can also be created to allow multiple selections. **List** provides these constructors:
List( ) throws HeadlessException
List(int *numRows*) throws HeadlessException
List(int *numRows*, boolean *multipleSelect*) throws HeadlessException
The first version creates a **List** control that allows only one item to be selected at any one time. In the second form, the value of *numRows* specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed). In the third form, if *multipleSelec*t is **true**, then the user may select two or more items at a time. If it is **false**, then only one item may be selected.
To add a selection to the list, call **add( )**. It has the following two forms:
void add(String *name*)
void add(String *name*, int *index*)
Here, *name* is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by *index*. Indexing begins at zero. If the value specified is –1 the item gets added to the end of the list.
For lists that allow only single selection, you can determine which item is currently selected by calling either **getSelectedItem( )** or **getSelectedIndex( )**. These methods are shown here:
String getSelectedItem( )
int getSelectedIndex( )
The **getSelectedItem( )** method returns a string containing the name of the item. If more than one item is selected, or if no selection has yet been made, **null** is returned. **getSelectedIndex( )** returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, –1 is returned.
For lists that allow multiple selection, you must use either **getSelectedItems( )** or **getSelectedIndexes( )**, shown here, to determine the current selections:
String[ ] getSelectedItems( )
int[ ] getSelectedIndexes( )
getSelectedItems( ) returns an array containing the names of the currently selected items. **getSelectedIndexes( )** returns an array containing the indexes of the currently selected items.
To obtain the number of items in the list, call **getItemCount( )**. You can set the currently selected item by using the **select( )** method with a zero-based integer index. These methods are shown here:
int getItemCount( )
void select(int *index*)
Given an index, the name associated with the item at that index can be obtained by calling **getItem()**, which has this general form:
String getItem(int *index*)
Here, *index* specifies the index of the desired item.

Managing Scroll Bars
Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow. The current value of the scroll bar relative to its minimum and maximum values is indicated by the *slider box* (or *thumb*) for the

scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this action translates into some form of page up and page down. Scroll bars are encapsulated by the **Scrollbar** class.

Scrollbar defines the following constructors:

Scrollbar( ) throws HeadlessException

Scrollbar(int *style*) throws HeadlessException

Scrollbar(int *style*, int *initialValue*, int *thumbSize*, int *min*, int *max*) throws HeadlessException

The first form creates a vertical scroll bar. The second and third forms allow you to specify the orientation of the scroll bar. If *style* is **Scrollbar.VERTICAL**, a vertical scroll bar is created. If *style* is **Scrollbar.HORIZONTAL**, the scroll bar is horizontal. In the third form of the constructor, the initial value of the scroll bar is passed in *initialValue.* The number of units represented by the height of the thumb is passed in *thumbSize.* The minimum and maximum values for the scroll bar are specified by *min* and *max.*

The minimum and maximum values of the scroll bar can be retrieved via **getMinimum( )** and

getMaximum( ), shown here:

int getMinimum( )

int getMaximum( )

By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line. The increment can be changed by calling **setUnitIncrement( )**.

By default, page-up and page-down increments are 10. This value can be changed by calling **setBlockIncrement( )**. These methods are shown here:

void setUnitIncrement(int *newIncr*)

void setBlockIncrement(int *newIncr*)

Handling Scroll Bars

To process scroll bar events, the **AdjustmentListener** interface is to be implemented. Each time a user interacts with a scroll bar, an **AdjustmentEvent** object is generated. Its **getAdjustmentType( )** method can be used to determine the type of the adjustment. The types of adjustment events are as follows:

- BLOCK_DECREMENT A page-down event has been generated.
- BLOCK_INCREMENT A page-up event has been generated.
- TRACK An absolute tracking event has been generated.
- UNIT_DECREMENT The line-down button in a scroll bar has been pressed.
- UNIT_INCREMENT The line-up button in a scroll bar has been pressed.

Using a TextField

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

TextField( ) throws HeadlessException

TextField(int *numChars*) throws HeadlessException

TextField(String *str*) throws HeadlessException
TextField(String *str*, int *numChars*) throws HeadlessException

The first version creates a default text field. The second form creates a text field that is *numChars* characters wide. The third form initializes the text field with the string contained in *str*. The fourth form initializes a text field and sets its width. **TextField** (and its superclass **TextComponent**) provides several methods that allow utilizing a text field. To obtain the string currently contained in the text field, **getText( )**can be called. To set the text, **setText( )** can be called. These methods are as follows:

String getText( )
void setText(String *str*)

Here, *str* is the new string. The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using **select( )**. Your program can obtain the currently selected text by calling **getSelectedText( )**. These methods are shown here:
String getSelectedText( )
void select(int *startIndex*, int *endIndex*)
the editability of a textfield cab be determined by calling **isEditable( )**. These methods are shown here:

boolean isEditable( )
void setEditable(boolean *canEdit*)

isEditable( ) returns **true** if the text may be changed and **false** if not. In **setEditable( )**, if *canEdit* is **true**, the text may be changed. If it is **false**, the text cannot be altered. When we don't want the text to be displayed, such as a password, the echoing of the characters as they are typed can be disabled by calling **setEchoChar( )**. This method specifies a single character that the **TextField** will display when characters are entered (thus, the actual characters typed will not be shown). The echo character can be retrieved by calling the **getEchoChar( )** method. These methods are as follows:
void setEchoChar(char *ch*)
boolean echoCharIsSet( )
char getEchoChar( )
Here, *ch* specifies the character to be echoed.

Using a TextArea
Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called **TextArea**. Following are the constructors for this**:**
TextArea( ) throws HeadlessException
TextArea(int *numLines*, int *numChars*) throws HeadlessException
TextArea(String *str*) throws HeadlessException
TextArea(String *str*, int *numLines*, int *numChars*) throws HeadlessException
TextArea(String *str*, int *numLines*, int *numChars*, int *sBars*) throws HeadlessException

Here, *numLines* specifies the height, in lines, of the text area, and *numChars* specifies its width, in characters. Initial text can be specified by *str*. In the fifth form, we can specify the scroll *sBars* must be one of these values
SCROLLBARS_BOTH SCROLLBARS_NONE
SCROLLBARS_HORIZONTAL_ONLY SCROLLBARS_VERTICAL_ONLY
TextArea is a subclass of **TextComponent**. Therefore, it supports the **getText( )**, **setText( )**, **getSelectedText( )**, **select( )**, **isEditable( )**, and **setEditable( )** methods described earlier.
TextArea adds the following methods:
void append(String *str*)
void insert(String *str*, int *index*)
void replaceRange(String *str*, int *startIndex*, int *endIndex*)
The **append( )** method appends the string specified by *str* to the end of the current text. **insert( )** inserts the string passed in *str* at the specified index. To replace text, call **replaceRange( )**. It replaces the characters from *startIndex* to *endIndex*–1, with the replacement text passed in *str*.

Understanding Layout Managers
A layout manager automatically arranges the controls within a window by using some type of algorithm. It is very tedious to manually lay out a large number of components and in some cases, the width and height information is not available when arranging the controls, Each **Container** object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout( )** method. If no call to **setLayout( )** is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it. The **setLayout( )** method has the following general form:
void setLayout(LayoutManager *layoutObj*)
Here, *layoutObj* is a reference to the desired layout manager.
Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time when a component is added to a container.

FlowLayout
FlowLayout is the default layout manager. **FlowLayout** implements a simple layout style, which is similar to how words flow in a text editor. The direction of the layout is governed by the container's component orientation property, which, by default, is left to right, top to bottom. Therefore, by default, components are laid out line-by-line beginning at the upper-left corner. In all cases, when a line is filled, layout advances to the next line. A small space is left between each component, above and below, as well as left and right. Here are the constructors for
FlowLayout:
FlowLayout( )
FlowLayout(int *how*)
FlowLayout(int *how*, int *horz*, int *vert*)

The first form creates the default layout, which centers components and leaves five pixels of space between each component. The second form lets you specify how each line is aligned.

Valid values for *how* are as follows:
FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT
FlowLayout.LEADING
FlowLayout.TRAILING

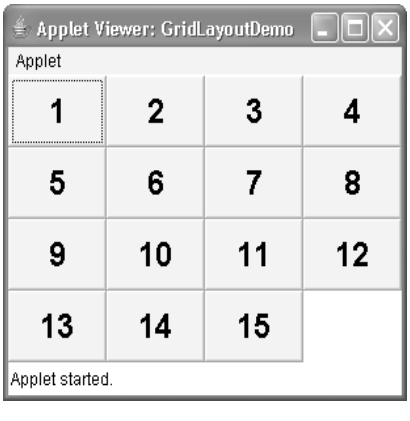
BorderLayout

The **BorderLayout** class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center. Here are the constructors defined
by BorderLayout:
BorderLayout( )
BorderLayout(int horz, int vert)
The first form creates a default border layout. The second allows you to specify the horizontal and vertical space left between components in horz and vert, respectively. BorderLayout defines the following constants that specify the regions:
BorderLayout.CENTER BorderLayout.SOUTH
BorderLayout.EAST BorderLayout.WEST
BorderLayout.NORTH
When adding components, we can specify the regions with **add( )**, which is defined by **Container**:
void add(Component *compObj*, Object *region*)
Here, *compObj* is the component to be added, and *region* specifies where the component will be added

GridLayout

GridLayout lays out components in a two-dimensional grid. When we instantiate a GridLayout, we define the number of rows and columns. The constructors supported by GridLayout are shown here:
GridLayout( )
GridLayout(int *numRows*, int *numColumns*)
GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*)
The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns. The third form allows to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimited-length columns. Specifying *numColumns* as zero allows for unlimited-length rows.

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



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

Menu Bars and Menus

Atop-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This concept is implemented in the AWT by the following classes: **MenuBar**, **Menu**, and **MenuItem**. In general, a menu bar contains one or more **Menu** objects. Each **Menu** object contains a list of **MenuItem** objects. Each **MenuItem** object represents something that can be selected by the user. Since **Menu** is a subclass of **MenuItem**, a hierarchy of nested submenus can be created.

To create a menu bar, first create an instance of **MenuBar**. This class only defines the default constructor. Next, create instances of **Menu** that will define the selections displayed on the bar. Following are the constructors for **Menu**:

Menu( ) throws HeadlessException

Menu(String *optionName*) throws HeadlessException

Menu(String *optionName*, boolean *removable*) throws HeadlessException

Here, *optionName* specifies the name of the menu selection. If *removable* is **true**, the menu can be removed and allowed to float free. Otherwise, it will remain attached to the menu bar. (Removable menus are implementation-dependent.) The first form creates an empty menu.

Individual menu items are of type **MenuItem**.

It defines these constructors:

MenuItem( ) throws HeadlessException

MenuItem(String *itemName*) throws HeadlessException

MenuItem(String *itemName*, MenuShortcut *keyAccel*) throws HeadlessException

Here, *itemName* is the name shown in the menu, and *keyAccel* is the menu shortcut for this item.

A menu item can be enabled or disabled by using the **setEnabled( )** method. Its form is shown here:

void setEnabled(boolean *enabledFlag*)

If the argument *enabledFlag* is **true**, the menu item is enabled. If **false**, the menu item is disabled.

The status on an item is obtained by calling **isEnabled( )**.

isEnabled( ) returns **true** if the menu item on which it is called is enabled. Otherwise, it returns **false**.

The name of a menu item can be changed by calling **setLabel( )**. The current name of the menu item can be obtained by using **getLabel( )**. These methods are as follows:

void setLabel(String *newName*)

String getLabel( )
Here, *newName* becomes the new name of the invoking menu item. **getLabel( )** returns the current name.

Menus only generate events when an item of type **MenuItem** is selected. They do not generate events when a menu bar is accessed to display a drop-down menu, for example. Each time a menu item is selected, an **ActionEvent** object is generated. By default, the action command string is the name of the menu item. However, you can specify a different action command string by calling **setActionCommand( )** on the menu item. Each time a check box menu item is checked or unchecked, an **ItemEvent** object is generated. Thus, you must implement the **ActionListener** and/or **ItemListener** interfaces in order to handle these menu events.

The **getItem( )** method of **ItemEvent** returns a reference to the item that generated this event. The general form of this method is shown here:
Object getItem( )

Introducing Swing

Swing was a response to deficiencies present in Java's original GUI subsystem: the Abstract Window Toolkit. The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface. One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or peers. This means that the look and feel of a component is defined by the platform, not by Java. Because the AWT components use native code resources, they are referred to as heavyweight.

Limitations of AWT Components

1. Because of variations between operating systems, a component might look, or even act, differently on different platforms. This potential variability threatened the overarching philosophy of Java: write once, run anywhere.
2. The look and feel of each component was fixed (because it is defined by the platform) and could not be (easily) changed.
3. The use of heavyweight components caused some frustrating restrictions. For example, a heavyweight component is always rectangular and opaque.

Advantages of Swing

Swing was created to address the limitations present in the AWT. It does this through two key features: lightweight components and a pluggable look and feel. Together they provide an elegant, yet easy-to-use solution to the problems of the AWT.

Swing Components Are Lightweight

Swing components are lightweight. This means that they are written entirely in Java and do not map directly to platform-specific peers. Lightweight components are rendered using graphics primitives; they can be transparent, which enables nonrectangular shapes. Thus, lightweight components are more efficient and more flexible. Because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system. This means that each component will work in a consistent manner across all platforms.

Swing Supports a Pluggable Look and Feel

Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing. This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does. Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects. In other words, it is possible to "plug in" a new look and feel for any given component without creating any side effects in the code that uses that component.

The pluggable look of swing is made possible because it uses the classic model-view-controller (MVC) architecture. In MVC terminology, the model corresponds to the state information associated with the component. For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked. The view determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model. The controller determines how the component reacts to the user. For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user's choice (checked or unchecked). This then results in the view being updated. By separating a component into a model, a view, and a controller, the specific implementation of each can be changed without affecting the other two. For instance, different view implementations can render the same component in different ways without affecting the model or the controller.

Although the MVC architecture and the principles behind it are conceptually sound, the high level of separation between the view and the controller is not beneficial for Swing components. Instead, Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the UI delegate. For this reason, Swing's approach is called either the Model-Delegate architecture or the Separable-Model architecture. Therefore, although Swing's component architecture is based on MVC, it does not use a classical implementation of it.

Components and Containers

Swing defines two types of items: components and containers. Component is an independent visual control, such as push button or text field. A container holds a group of components. If a component needs to be displayed, it must be held within a container. Because containers are also components, a container can also hold other containers. This enables Swing to define containment hierarchy.

Components

Swing components are derived from the JComponent class. JComponent provides the functionality that is common to all components. For example, JComponent supports the pluggable look and feel. JComponent inherits the AWT classes Container and Component. Thus, a Swing component is built on and compatible with an AWT component. All of Swing's components are represented by classes defined within the package javax.swing. The following table shows the class names for Swing components (including those used as containers).

JApplet JColorChooser	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayer
JLayeredPlane	JList	JMenu	JMenuBar
JMenuItem	JOptionPlane	JPlane	JPasswordField
JPopupMenu	JProgressBar	JRadioButton	JRadioButtonMenuItem
JRootPane	JScrollBar	JSCrollPane	JSeperator
JSlider	JSpinner	JSplitPane	JTabbedPane
JTable	JTextArea	JTextField	JtextPane
JToggleButton	JToolBar	JToolTip	JTree
JViewport	JWindow		

Containers

Swing defines two types of containers. The first are top-level containers: **JFrame**, **JApplet**, **JWindow**, and **JDialog**. These containers do not inherit **JComponent**. They do, however, inherit the AWT classes **Component** and **Container**. Unlike Swing's other components, which are lightweight, the top-level containers are heavyweight. This makes the top-level containers a special case in the Swing component library. As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained within any other container. Furthermore, every containment hierarchy must begin with a top-level container. The one most commonly used for applications is **JFrame**. The one used for applets is **JApplet**.

The second type of containers supported by Swing are lightweight containers. Lightweight containers do inherit **JComponent**. Examples of a lightweight container is **JPanel**, **JScrollPane** and **JRootPane** Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container. Thus lightweight containers such as **JPanel** can be used to create subgroups of related controls that are contained within an outer container.

Layout Managers

Layout manager controls the position of component within a container. Java offers several layout managers. Most are provided by AWT. Here is a list of a few layout managers available:

FlowLayout	A simple layout that positions components left-to-right, top-to-bottom
BorderLayout	Positions the components within the center or the borders of the container.
GridLayout	Lays out components within a grid
GirdBagLayout	Lays out different size components within a flexible grid
BoxLayout	Lays out components vertically or horizontally within a box
SpringLayout	Lays out components subject to a set of constraints

- BorderLayout is the default layout. It defines five locations to which a component can be added. They are center, north, south east and west. By default, it is center. To add a component to one of the regions, it is necessary to specify the name of the region.
- FlowLayout lays out components one row at a time, top to bottom. When one row is full, layout advances to the next row.

```
// A simple Swing application.
import javax.swing.*;
class SwingDemo {
SwingDemo() {
// Create a new JFrame container.
JFrame jfrm = new JFrame("A Simple Swing Application");
// Give the frame an initial size.
jfrm.setSize(275, 100);
// Terminate the program when the user closes the application.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// Create a text-based label.
JLabel jlab = new JLabel(" Swing means powerful GUIs.");
// Add the label to the content pane.
jfrm.add(jlab);
// Display the frame.
jfrm.setVisible(true); }
public static void main(String args[]) {
// Create the frame on the event dispatching thread.
SwingUtilities.invokeLater(new Runnable() {
public void run() {
new SwingDemo(); }
});
}  }
```

Explanation:
The **javax.swing** package contains the components and models defined by Swing. It defines classes that implement labels, buttons and other controls. This package will be included in all programs that use swing.
`JFrame jfrm = new JFrame("A Simple Swing Application");`
Will create a container called jfrm that defines a rectangular window, with title bar, maximize, minimize and close buttons with title specified by the constructor.
`jfrm.setSize(275, 100);`
The setSize() method sets dimensions of the window, which are specified in pixels. Its general form is `void setSize(int width, int height);`
By default, when top-level window is closed, the window is removed from the screen but application is not terminated. One can make the application terminate by specifying setDefaultCloseOperation() which has the general format void setDefaultCloseOperation(int what). The value passed in what determines what happens when the window is closed. There are several options:
JFrame.DESPOSE_ON_CLOSE, JFrame.HIDE_ON_CLOSE, JFrame.EXIT_ON_CLOSE, JFrame.DO_NOTHING_ON_CLOSE are some of the options.

Use of JButton

One of the most commonly used Swing controls is the push button. It is an instance of JButton. Swing push buttons can contain text, image or both. JButton supplies several constructors. The one which is most commonly used is

`JButton``(String msg)` The msg specifies the string that will be displayed inside the button.

When push button is pressed, it generates ActionEvent which is defined by AWT and also Swing. JButton provides the following methods, which are used to add or remove an action listener:

`JButton.addActionListener(ActionListener al);`

`JButton.removeActionListener(ActionListener al)` Here *al* specifies an object that will receive an instance of a class that implements ActionListener interface. The ActionListener interface defines actionPerformed() method. This method is called when a button is pressed.

Example:
```
JButton reset = new JButton("Reset");
    reset.addActionListener(this);
```
When we add more controls to a container, we must set the layout manager for the content pane as follows:

```
jfrm.setLayout(new FlowLayout());
```

JTextField

It is also a commonly used control. It defines several constructors. The most commonly used one is

`JTextField(int cols)` Here, cols specifies the width of text field in columns. A text which is longer than columns can be entered because cols specifies the physical size of the text field on screen. It is used for accepting input from the user. When accepting input, if enter key is pressed, an ActionEvent is generated. Therefore JTextField provides addActionListener() and removeActionListener() methods. To handle action events, actionPerformed() method defined by ActionListener interface must be implemented.

JTextField has an action command string associated with it. By default, the action command is the current content of the text field. We can set action command to a fixed value by calling setActionCommand() method which has a general format

```
void setActionCommand(String cmd)
```

The string passed in cmd becomes the new action command string. The text in the text field remains unaffected.

To obtain the string that is currently displayed in the text field, getText() method can be called. To set the contents to a specified value setText() method can be used.

Example:

`JTextField jtf= new JTextField(10);` -- defines a text field which is 10 columns wide.

`jtf.setActionCommand("myTF");` - This will make myTF as action command when event is generated.

`jtf.setText("Use text field to input");` - This will set the contents of jtf as Use text field to input

`String str= jtf.getText();` – Whatever is stored in text field jtf will be transferred to str which is of type string.

JCheckBox

In swing, a check box is a special type of button which belongs to the type JCheckBox and inherits the AbstractButton and ToggleButton classes. Check box defines several constructors. The commonly used one is

`JCheckBox(String str)`

It creates a check box that has text specified by str as label. When the user selects or deselects a check box, an **ItemEvent** is generated. You can obtain a reference to the **JCheckBox** that generated the event by calling **getItem( )** on the **ItemEvent** passed to the **itemStateChanged( )** method defined by **ItemListener**. This interface specifies only one method, **itemStateChanged()**.

void itemStateChanged(ItemEvent ie)

The item event received is ie. To obtain reference to the item that is changed, getItem() can be used on the ItemEvent object. This has the following syntax:

Object getItem() → the reference returned must cast to the component class being handled. In this case JCheckBox.

The getText() and setText() methods can be used to obtain and set the contents of the check box.

The easiest way to determine the state of a check box is by calleing isSelected() method. It will return true, if the check box is selected else it will return false. It has the following syntax:

boolean isSelected()

Example:

`JCheckBox jcb = new JCheckBox("Alpha");` this creates a check box named jcb with label Alpha

`jcb.addItemListener(this);` this will set action to the check box jcb. When this check box is checked, itemEvent is generated whci can be caught as follows;

```
public void itemStateChanged(ItemEvent ie)
{JCheckBox cb = (JCheckBox) ie.getItem();
if(cb.isSelected())
jtf.setText(cb.getText() + "Is selected");
else
jtf.setText( " check box is not selected"); }
```

JList

In Swing, the basic list class is called **JList**. It supports the selection of one or more items from a list. Although the list often consists of strings, it is possible to create a list of just about any object that can be displayed inside a **JScrollPane**. This way, long lists will automatically be scrollable, which simplifies GUI design. It also makes it easy to change the number of entries in a list without having to change the size of the **JList** component.

A **JList** generates a **ListSelectionEvent** when the user makes or changes a selection. This event is also generated when the user deselects an item. It is handled by implementing

ListSelectionListener. This listener specifies only one method, called **valueChanged( )**, which is shown here:
void valueChanged(ListSelectionEvent *le*)
Here, *le* is a reference to the object that generated the event. Although **ListSelectionEvent** does provide some methods of its own, normally we will interrogate the **JList** object itself to determine what has occurred. Both **ListSelectionEvent** and **ListSelectionListener** are packaged in **javax.swing.event**.

By default, a **JList** allows the user to select multiple ranges of items within the list, but we can change this behavior by calling **setSelectionMode( )**, which is defined by **JList**. It is shown here:
void setSelectionMode(int *mode*)
Here, *mode* specifies the selection mode. It must be one of these values defined by **ListSelectionModel** interface of the package javax.swing.
SINGLE_SELECTION, SINGLE_INTERVAL_SELECTION, MULTIPLE_INTERVAL_SELECTION
The default, multiple-interval selection, lets the user select multiple ranges of items within a list. With single-interval selection, the user can select one range of items. With single selection, the user can select only a single item. Of course, a single item can be selected in the other two modes, too. It's just that they also allow a range to be selected.
The index of the first item selected can be obtained, which will also be the index of the only selected item when using single-selection mode, by calling **getSelectedIndex( )**, shown here:
int getSelectedIndex( )
Indexing begins at zero. So, if the first item is selected, this method will return 0. If no item is selected, –1 is returned

Use of Anonymous Inner Classes to Handle Events
Till now, event handling was straight forward, in which the main class of application will implement the listener interface itself and all events are sent to an instance of that class. We can also implement listeners through the use of anonymous inner classes. Anonymous inner classes are inner classes that don't have a name. Instead, an instance of the class is generated on the fly as needed. Anonymous inner classes make implementing some types of event handlers much easier. For example, when the JBtton named jbtn is clicked, the action listener can be implemented as follows:

```
Jbtn.addActionListenerr(new ActionerListener() {
public void actionPerformed(ActionEvent ae)
 {  // code to handle action event;} } );
```

Here anonymous inner class is created that implements the ActionListener interface. One advantage of using anonymous inner class is that the component that invokes the method is already known. In the example above there is no need to call getActionCommand() to determine which component generated the event, because this implementation of ActionPerformed() will only be called by events generated by jbtn.

Create a Swing Applet

The second type of applications that commonly use Swing are Swing applets. They are similar to AWT-based applets. They extend JApplet class. Swing applets use the four life cycle methods init(), start() stop() and destroy(). JApplet will not override the paint() method. All interactions with components in a Swing applet take place on the event-dispatching thread.

```
// A simple Swing-based applet
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
/*
This HTML can be used to launch the applet:
<object code="MySwingApplet" width=220 height=90>
</object>
*/
public class MySwingApplet extends JApplet {
JButton jbtnAlpha;
JButton jbtnBeta;
JLabel jlab;
// Initialize the applet.
public void init() {
try {
SwingUtilities.invokeAndWait(new Runnable () {
public void run() {
makeGUI(); // initialize the GUI
}
});
} catch(Exception exc) {
System.out.println("Can't create because of "+ exc);
}
}
// This applet does not need to override start(), stop(),
// or destroy().
// Set up and initialize the GUI.
private void makeGUI() {
// Set the applet to use flow layout.
setLayout(new FlowLayout());
// Make two buttons.
jbtnAlpha = new JButton("Alpha");
jbtnBeta = new JButton("Beta");
// Add action listener for Alpha.
jbtnAlpha.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent le) {
jlab.setText("Alpha was pressed.");
}
});
// Add action listener for Beta.
```

```
jbtnBeta.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent le) {
jlab.setText("Beta was pressed.");
}
});
// Add the buttons to the content pane.
add(jbtnAlpha);
add(jbtnBeta);
// Create a text-based label.
jlab = new JLabel("Press a button.");
// Add the label to the content pane.
add(jlab);
}
}
```