Chapter-8

APPLET, EVENT HANDLING, SWINGS:

8.1 Applet:

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

A simple applet program:

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

Program explanation:

- Applet must be declared as public, because it will be accessed by code that is outside the program.
- Inside SimpleApplet, paint( ) is declared.
- This method is defined by the AWT and must be overridden by the applet.
- paint( ) is called each time that the applet must redisplay its output.

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

Two ways in which you can run an applet:

- Executing the applet within a Java-compatible web browser.
- Using an applet viewer, such as the standard tool, appletviewer.

- An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.
- To execute an applet in a web browser, you need to write a short HTML text file that contains a tag that loads the applet.
- Currently, Sun recommends using the APPLET tag for this purpose.
- Here is the HTML file that executes SimpleApplet:

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

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

Faster way of executing an applet :

- Include the applet code in your program itself

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

Execution procedure:
1) Compile your program.
2) Execution :
C:\AppletViewer programname.java

Two Types of Applets

There are two varieties of applets

1)Applet:-These applets use the Abstract Window Toolkit(AWT) to provide the graphic user interface

2)JApplet:-The second type of applet is JApplet are those based on the Swing class.Swing applet use the Swing classes to provide the GUI.
Because JApplet inherits Applet ,all the features of Applet are also available in JApplet .

Applet Basics

- All applets are subclasses (either directly or indirectly) of Applet.
- Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer.
- Execution of an applet does not begin at main().
- Output to your applet’s window is not performed by System.out.println.
- Rather, in non-Swing applets, output is handled with various AWT methods, such as drawString(),Which outputs a string to a specified X,Y location.

The Applet Class:

- Applet provides all necessary support for applet execution, such as starting and stopping.
- It also provides methods that load and display images, and methods that load and play audio clips.
- Applet extends the AWT class Panel.
- In turn, Panel extends Container, which extends Component.
- Thus, Applet provides all of the necessary support for window-based activities.

Method	**Description**
void destroy( )	Called by the browser just before an applet is terminated. Your applet will override this method if it needs to perform any cleanup prior to its destruction.
AudioClip getAudioClip(URL url)	Returns an AudioClip object that encapsulates the audio clip found at the location specified by url.
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.
String getParameter(String paramName)	Returns the parameter associated with paramName. null is returned if the specified parameter is not found.
void init( )	Called when an applet begins execution. It is the first method called for any applet.
boolean isActive( )	Returns true if the applet has been started. It returns false if the applet has been stopped.
void play(URL url)	If an audio clip is found at the location specified by url, the clip is played.
void play(URL url, String clipName)	If an audio clip is found at the location specified by url with the name specified by clipName, the clip is played.
void resize(int width, int height)	Resizes the applet according to the dimensions specified by width and height.

Applet Architecture

- An applet is a window-based program. As such, its architecture is different from the so called Normal, console-based programs.

  There are a few key concepts you must understand:
- First, applets are event driven. 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.
- Second, the user initiates interaction with an applet.
- 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.

An Applet Skeleton

- All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution.
- Four of these methods are → ***init( ), start( ), stop( ), and destroy( ) these are defined by Applet***.
- *Another*, ***paint( ), is defined by the AWT Component class***. Applets do not need to override those methods because they do not use.

Here is an example of Applet program with all basic functions

```
import java.awt.*;
import java.applet.*;
/* <applet code="AppletTest1" width=200 height=200>
</applet> */
public class AppletTest1 extends Applet
{
 public void init() {
  System.out.println("inside init");
 }
 public void start(){
  System.out.println("inside start");
 }
```

```
public void stop(){
  System.out.println("inside Stop");
 }
 public void destroy() {
  System.out.println("inside destroy");
 }
 public void paint(Graphics g)
 {
   g.drawString("Hello Applet", 20,20);
 }
}
```

Applet Initialization and Termination

- **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( ). 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.
- **paint( ) :-** The paint( ) method is called each time your applet's output must be redrawn.
- **stop( ) :-** The stop( ) method is called when a web browser leaves the HTML document containing the applet when it goes to another page,
- **destroy( ) :-** The destroy( ) method is called when the environment determines that your applet needs to be removed completely from memory.

Simple Applet Display Methods

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

- To set the background color of an applet's window, use **setBackground( )**.
- To set the foreground color (the color in which text is shown, for example), use **setForeground( )**.

These methods are defined by Component, and they have the following general forms:

```
void setBackground(Color newColor)
void setForeground(Color newColor )
```

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;
public void init() {
setBackground(Color.cyan);
setForeground(Color.red);
msg = "Inside init( ) --";
}
```

```
public void start() {
msg += " Inside start( ) --";
}
// Display msg in applet window.
public void paint(Graphics g) {
msg += " Inside paint( ).";
g.drawString(msg, 10, 30);
}
}
```

Requesting Repainting

- Whenever your applet needs to update the information displayed in its window, it simply calls **repaint( )**.
- 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( ).
- For example, if part of your applet needs to output a string, it can store this string in a String variable and then call repaint( ).
- Inside paint( ), you will output the string using drawString( ).
- 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.
- 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.

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.

A Simple Banner Applet

- To demonstrate repaint( ), a simple banner applet is developed.
- This applet scrolls a message, from right to left, across the applet’s window.
- Since the scrolling of the message is a repetitive task, it is performed by a separate thread, created by the applet when it is initialized.

The banner applet is shown here:

(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;

// 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) {}
}// end of for
}  // end of run
// Pause the banner.
public void stop() {
stopFlag = true;
t = null;
}
// Display the banner.
public void paint(Graphics g) {
g.drawString(msg, 50, 30);
}}
```

- First, notice that SimpleBanner extends Applet, as expected, but it also implements Runnable.
- The AWT run-time system calls start( ) to start the applet running.
- Inside start( ), a new thread of execution is created and assigned to the Thread variable t.
- Then, the boolean variable stopFlag, which controls the execution of the applet, is set to false.
- Next, the thread is started by a call to t.start( ).

- Remember that t.start( ) calls a method defined by Thread, which causes run( ) to begin executing.
- It does not cause a call to the version of start( ) defined by Applet.
- These are two separate methods.Inside run( ), the characters in the string contained in msg are repeatedly rotated left.
- Between each rotation, a call to repaint( ) is made. This eventually causes the paint( ) method to be called and the current contents of msg is displayed.
- Between each iteration, run( ) sleeps for a quarter of a second.
- The net effect of run( ) is that the contents of msg is scrolled right to left in a constantly moving display.
- The stopFlag variable is checked on each iteration. When it is true, the run( ) method terminates.

Using the Status 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 following applet demonstrates **showStatus( )**:

```
// 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.");
}
}
```

The HTML APPLET Tag

- The APPLET tag is used to start an applet from both an HTML document and from an applet viewer.
- An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers like Netscape Navigator, Internet Explorer, and HotJava will allow many applets on a single page.

The syntax for the standard APPLET tag is shown here.

```
< 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>
```

Let's take a look at each part now:

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).

CODE:- CODE is a required attribute that gives the name of the file containing your applet's compiled **.class** file.

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.

NAME:- NAME is an optional attribute used to specify a name for the applet instance. 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.

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.

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.

Passing Parameters to Applets

- To retrieve a parameter, use the getParameter( ) method.
- It returns the value of the specified parameter in the form of a String object.

Here is an example that demonstrates passing parameters:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="PDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=accountEnabled value=true>
</applet> */
public class PDemo extends Applet{
String fontName;
int fontSize;
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("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("Account Active: " + active, 0, 58);
}
}
```

getDocumentBase( ) and getCodeBase( )

- You will create applets that will need to explicitly load media and text. Java will allow the applet to load data from the directory holding the HTML file that started the applet
- (The document base) and the directory from which the applet's class file was loaded (the code base).

A Simple Program showing document base

```
// Display code and document bases.
import java.awt.*;
import java.applet.*;
import java.net.*;
/* <applet code="Bases" width=300 height=50>
</applet> */
public class Bases extends Applet{
// Display code and document bases.
public void paint(Graphics g) {
String msg;
URL url = getCodeBase(); // get code base
msg = "Code base: " + url.toString();
g.drawString(msg, 10, 20);
url = getDocumentBase(); // get document base
msg = "Document base: " + url.toString();
g.drawString(msg, 10, 40);
}}
```

AppletContext and showDocument( )

- To allow your applet to transfer control to another URL, you must use the showDocument( ) method defined by the AppletContext interface.
- AppletContext is an interface that lets you get information from the applet's execution environment.
- The context of the currently executing applet is obtained by a call to the getAppletContext( ) method defined by Applet.
- Within an applet, once you have obtained the applet's context, you can bring another document into view by calling showDocument( ).
- There are two showDocument( ) methods. The method showDocument(URL) displays the document at the specified URL.
- The method showDocument(URL, where) displays the specified document at the specified location within the browser window.

8.2 **Event Handling:** (changing activity ex: mouse event or keyboard event):

- Event handling is integral to the creation of applets and other types of GUI-based programs
- Without the event handling mechanism it is not possible to write the GUI codes.
- Events are supported by a number of packages, including java.util, java.awt, and java.awt.event.

- **The Delegation Event Model**: Delegation event model, which ***defines standard and consistent mechanisms to generate and process events***.

  Working of Delegation event model: 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 an event is received, the listener processes the event and then returns.

Events

- In the delegation model, an ***event is an object that describes a state change in a source***.
- ***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.
- ***Event may be generated when a timer expires, a counter exceeds a value, software or hardware failure occurs, or an operation is completed.***
- You are free to define events that are appropriate for your application.

1) Event Sources

- **A *source is an object that generates an event.***
- 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.

Multi casting: (Registering more than one event)

Here is the general form:

public void add*TypeListener(TypeListener el)* *(Multi casting means more than one 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.*

Unicasting (Registering only one event)

public void add*TypeListener(TypeListener el)* throws java.util.TooManyListenersException (unicasting)

Unregistering an 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*TypeListener(TypeListener el)*
- you would call **removeKeyListener( ).**

2) 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.
- For example, the ***MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved*.**

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.
- It's one constructor is shown here:
  constructor: **EventObject(Object *src)***
  Note: Here, *src is the object that generates this event.*

<u>EventObject contains two methods:</u>

getSource( ) and toString( ).

1) The getSource( ) method

- returns the source of the event. Its general form is shown here:

constructor: **Object getSource( )**

2) toString( ) returns the string equivalent of the event.

<u>AWTEvent:</u>

- 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.

NOTE:

<u>To summarize:</u>

• EventObject is a superclass of all events.

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

- Commonly used constructors and methods in each class are described in the following sections:

<u>The Action Event Class</u>

- An **Action Event** is generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
- The **Action Event** class defines four integer constants that can be used to identify any modifiers associated with an action event:
- **ALT_MASK, CTRL_MASK, META_MASK, and SHIFT_MASK.**
- In addition, there is an integer constant, **ACTION_ PERFORMED,** which can be used to identify action events.

<u>Classes in java.awt.event:</u>

Event Class	Description
Action Event	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.

The KeyEvent Class (used for keyboard events)

- A KeyEvent is generated when keyboard input occurs.
- There are three types of key events, which are identified by these integer constants:

  KEY_PRESSED, KEY_RELEASED, and KEY_TYPED.
- The first two events i.e KEY_PRESSED, KEY_RELEASED are generated when any key is pressed or released.
- The last event i.e. KEY_TYPED occurs only when a character is generated

  NOTE: Remember, not all key presses result in characters. For example, pressing SHIFT does not generate a character.
- There are many other integer constants that are defined by KeyEvent.
- For example, VK_0 through VK_9 and VK_A through VK_Z define the ASCII equivalents of the numbers and letters.

  Here are some others:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK.SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

- The VK constants specify virtual key codes and are independent of any modifiers, such as control, shift, or alt.

  KeyEvent class is a subclass of InputEvent

  Here is one of its constructors:

  KeyEvent(Component src, int type, long when, int modifiers, int code, char ch)
- 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 key was pressed is passed in when.
- The modifiers argument indicates which modifiers were pressed when this key event occurred. (ALT, SHIFT, CTRL)
- The virtual key code, such as VK_UP, VK_A, and so forth, is passed in code.
- The character equivalent (if one exists) is passed in ch.
- If no valid character exists, then ch contains CHAR_UNDEFINED.
- For KEY_TYPED events, code will contain VK_UNDEFINED.

KeyEvent class methods:

- ***The KeyEvent class defines several methods, but the most commonly used ones are getKeyChar( ), which returns the character that was entered, and getKeyCode( ), which returns the key code.***

  Their general forms are shown here:

  char getKeyChar( )

  int getKeyCode( )

- If no valid character is available, then **getKeyChar( )** returns **CHAR_UNDEFINED.**

When a **KEY_TYPED** event occurs when no valied key is pressed , **getKeyCode( )** returns **VK_UNDEFINED.**

Key board event program:

```
// 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);
}
```

```
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);
}
}
```

The MouseEvent Class (used for keyboard events):

- 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.

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)*

1. The coordinates of the mouse are passed in *x and y*.
2. The click count is passed in clicks.
3. The triggersPopup flag indicates if this event causes a pop-up menu to appear on this platform.

- Two commonly used methods in this class are **getX( ) and getY( ).** These return the X and Y coordinates of the mouse within the component 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 **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( )

Also available is 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

- Java SE 6 added three methods to MouseEvent that obtain the coordinates of the mouse relative to the screen rather than the component.

  They are shown here:

  1. Point getLocationOnScreen( )
  2. int getXOnScreen( )
  3. int getYOnScreen( )

Sources of Events: these are the classes which can be used in the place of src.

Event Source	**Description**
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

- **Event Listener Interfaces**
- **The ActionListener Interface**

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.

The KeyListener Interface

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

The MouseListener Interface:

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

The general forms of these methods are shown here:

1. void mouseClicked(MouseEvent *me)*
2. void mouseEntered(MouseEvent *me)*
3. void mouseExited(MouseEvent *me)*
4. void mousePressed(MouseEvent *me)*
5. void mouseReleased(MouseEvent *me)*

The MouseMotionListener Interface

- This interface defines two methods. The **mouseDragged( ) method is called multiple times** as the mouse is dragged.
- The **mouseMoved( ) method is called multiple times as the mouse** is moved.

  Their general forms are shown here:

  1. void mouseDragged(MouseEvent *me)*
  2. void mouseMoved(MouseEvent *me)*

Using the Delegation Event Model:

- Now that you have learned the theory behind the delegation event model and have had an overview of its various components, it is time to see it in practice.

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.**

Handling Mouse Events:

- To handle mouse events, you must implement the **MouseListener and the MouseMotionListener** interfaces.

Working of the mouse event Applet program:

- It displays the current coordinates of the mouse in the applet's status window.
- Each time a button is pressed, the word "Down" is displayed at the location of the mouse pointer.
- Each time the button is released, the word "Up" is shown.
- If a button is clicked, the message "Mouse clicked" is displayed in the upperleft corner of the applet display area.
- As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area.
- When dragging the mouse, a * is shown, which tracks with the mouse pointer as it is dragged.
- Notice that the two variables, **mouseX and mouseY, store** the location of the mouse when a mouse pressed, released, or dragged event occurs.
- These coordinates are then used by **paint( ) to display output at the point of these occurrences.**

Working of the mouse event Applet program:

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

public void mouseEntered(MouseEvent me) {}
```

```
public void mouseExited(MouseEvent me) {}

public void mousePressed(MouseEvent me) {}

public void mouseReleased(MouseEvent me) {
}

public void mouseDragged(MouseEvent me) {
}

public void mouseMoved(MouseEvent me) {
}
public void paint(Graphics g) {
g.drawString(msg, mouseX, mouseY);
}}
```

Note: It is compulsory to define all the above functions as we are implementing the interfaces (MouseMotionsListener & MouseListener).
But it is **not compulsory** to write some functionality in the body, we can keep the body of the functions as empty, as we had done here.

Explanation of program :

- The **MouseEvents class extends Applet and implements** both the **MouseListener and MouseMotionListener interfaces.**
- These two interfaces contain methods that receive and process the various types of mouse events.
- Notice that the applet is both the source and the listener for these events.
- This works because **Component,** which supplies the **addMouseListener( ) and addMouseMotionListener( )** methods, is a superclass of **Applet.**
- Being both the source and the listener for events is a common situation for applets.
- Inside **init( ),** the applet registers itself as a listener for mouse events.
- This is done by using **addMouseListener( ) and addMouseMotionListener( ),** which, as mentioned, are members of Component.

They are shown here:

void addMouseListener(MouseListener *ml)*

void addMouseMotionListener(MouseMotionListener *mml)*

addMouseListener(this);

addMouseMotionListener(this)

- The two methods are the part of the Component class which is used to register the Mouse events.

  The Original Methods of Component class are:

  void addMouseListener(MouseListener *ml)*

  void addMouseMotionListener(MouseMotionListener *mml)*

Adapter Classes:

- 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( ),** which are the methods defined by the **MouseMotionListener** interface.
- If you were interested in only mouse drag events, then you could simply extend **MouseMotionAdapter** and **override mouseDragged( ).**
- The empty implementation of **mouseMoved( )** would handle the mouse motion events for you.

Adapter Classes:

- Commonly Used Listener Interfaces Implemented by Adapter Classes

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

- 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 extends MouseAdapter and overrides the mouseClicked( ) method.**
- The other mouse events are silently ignored by code inherited from the **MouseAdapter** class.
- **MyMouseMotionAdapter extends MouseMotionAdapter and overrides the mouseDragged( ) method.**
- The other mouse motion event is silently ignored by code inherited from the **MouseMotionAdapter class.**

<u>Program below here shows the example of an adapter class:</u>

```
// 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 AD_obj;
public MyMouseAdapter(AdapterDemo obj)
{
AD_obj= obj
}
```

```
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
AD_obj.showStatus("Mouse clicked");
}
}

class MyMouseMotionAdapter extends
MouseMotionAdapter {
AdapterDemo AD_obj;
public
MyMouseMotionAdapter(AdapterDemo obj)
{
AD_obj= obj
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me)
{
AD_obj.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.**

Program below Does not uses the Inner Class

```
// 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.");
}
}
```

Program showing the use of Inner classes:

```
// Inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="InnerClassDemo" width=200
height=100>
</applet>
*/
public class InnerClassDemo extends Applet {
```

```
public void init() {
addMouseListener(new MyMouseAdapter());
}
class MyMouseAdapter extends MouseAdapter {
public void mousePressed(MouseEvent me) {
showStatus("Mouse Pressed");
}
}
}
```

Anonymous Inner Classes:

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

```
// 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"); }} );
}
}
```

8.3 Swings:

- Although the AWT is still a crucial part of Java, its component set is no longer widely used to create graphical user interfaces.
- Today, most programmers use Swing for this purpose.
- Swing is a set of classes that provides more powerful and flexible GUI components than does the AWT.
- Simply put, Swing provides the look and feel of the modern Java GUI.

Two Key Swing Features:

- Two key features: **lightweight components** and a **pluggable look and feel**.

Swing Components Are Lightweight:

- With very few exceptions, Swing components are *lightweight. This means that they are written* entirely in Java and do not map directly to platform-specific peers.

Components and Containers:

- ***A component is an independent visual control, such as a push button or slider***.
- ***A container holds a group of components***.
- Thus, a container is a special type of component that is designed to hold other components.
- Furthermore, in order for a component to be displayed, it must be held within a container.
- Thus, all Swing GUIs will have at least one container.
- Because containers are components, a container can also hold other containers. This enables Swing to define what is called a *containment hierarchy, at the top of which must be a top-level container.*

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.**
- All of Swing's components are represented by classes defined within the package **javax.swing.**

Components:

Table shows the class names for Swing components (including those used as containers).

JApplet	J Button	JCheckBox	JCheckBoxMenultem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextReld
J Frame	JlnternalFrame	JLabel	JLayeredPane
JList	JMenu	JMenuBar	JMenultem
JOptionPane	JPanel	JPasswordField	JPopupMenu
JProgressBar	JRadioButton	JRadioButtonMenultem	JRootPane
JScrollBar	JScrollPane	JSeparator	JSlider
JSpinner	JSplitPane	JTabbedPane	JTable
JTextArea	JTextReld	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.
- 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.**
- An example of a lightweight container is **JPanel,** which is a general-purpose container.

The Top-Level Container Panes:

- Each top-level container defines a set of *panes. At the top of the hierarchy is an instance of* **JRootPane.**
- **JRootPane** is a lightweight container whose purpose is to manage the other panes.
- It also helps manage the optional menu bar. The panes that comprise the root pane are called the glass pane, the content pane, and the layered pane.

The Swing Packages

javax.swing	javax.swing.border	javax.swing.colorchooser
javax.swing.event	javax.swing.filechooser	javax.swing.plaf
javax.swing.plaf.basic	javax.swing.plaf.metal	javax.swing.plaf.multi
javax.swing.plaf.synth	javax.swing.table	javax.swing.text
javax.swing.text.html	javax.swing.text.html.parser	javax.swing.text.rtf
javax.swing.tree	javax.swing.undo	

A Simple Swing Application:

- Swing programs differ from both the console-based programs and the AWT-based programs they use a different set of components and a different container hierarchy than does the AWT.

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

A Simple Swing Application Explanation:

- It begins by creating a JFrame, using this line of code:

  JFrame jfrm = new JFrame("A Simple Swing Application");

- This creates a container called jfrm that defines a rectangular window complete with a title bar; close, minimize, maximize, and restore buttons; and a system menu.

  Next, the window is sized using this statement:

  jfrm.setSize(275, 100);

- The **setSize( )** method (which is inherited by JFrame from the AWT class Component) sets the dimensions of the window, which are specified in pixels.

  Its general form is shown here:

  void setSize(int *width, int height)*

setDefaultCloseOperation( )

As the Program Uses this Line:

jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

- By default, when a top-level window is closed (such as when the user clicks the close box), the window is removed from the screen, but the application is not terminated.
- While this default behavior is useful in some situations, it is not what is needed for most applications.
- Instead, you will usually want the entire application to terminate when its top-level window is closed.
- There are a couple of ways to achieve this. The easiest way is to call

  setDefaultCloseOperation( ),

- general form of **setDefaultCloseOperation( )** is shown here:

  void setDefaultCloseOperation(int *what)*

- The value passed in what determines what happens when the window is closed.
- There are several other options in addition to **JFrame.EXIT_ON_CLOSE.**

  They are shown here:

  JFrame.DISPOSE_ON_CLOSE

  JFrame.HIDE_ON_CLOSE

  JFrame.DO_NOTHING_ON_CLOSE

  jfrm.add(jlab);

General form of add( ) is shown here:

Component add(Component *comp);*

jfrm.setVisible(true);

- The **setVisible( )** method is inherited from the AWT Component class. If its argument is true, the window will be displayed.
- Otherwise, it will be hidden.
- By default, a JFrame is invisible, so **setVisible(true)** must be called to show it.
- Inside main( ), a SwingDemo object is created, which causes the window and the label to be displayed.
- Notice that the SwingDemo constructor is invoked using these lines of code:

SwingUtilities.invokeLater(new Runnable(){ public void run() { new SwingDemo(); }});

- This sequence causes a SwingDemo object to be created on the event dispatching thread rather than on the main thread of the application.
- Here's why. In general, Swing programs are event-driven.
- For example, when a user interacts with a component, an event is generated.
- An event is passed to the application by calling an event handler defined by the application. However, the handler is executed on the event dispatching thread provided by Swing and not on the main thread of the application.
- Thus, although event handlers are defined by your program, they are called on a thread that was not created by your program.
- To avoid problems (including the potential for deadlock), all Swing GUI components must be created and updated from the event dispatching thread, not the main thread of the application.
- However, main( ) is executed on the main thread.
- Thus, main( ) cannot directly instantiate a SwingDemo object.
- Instead, it must create a Runnable object that executes on the event dispatching thread and have this object create the GUI.
- To enable the GUI code to be created on the event dispatching thread, you must use one of two methods that are defined by the SwingUtilities class.
- These methods are invokeLater( ) and invokeAndWait( ).

They are shown here:

static void invokeLater(Runnable *obj*)

static void invokeAndWait(Runnable *obj*)throws InterruptedException, InvocationTargetException

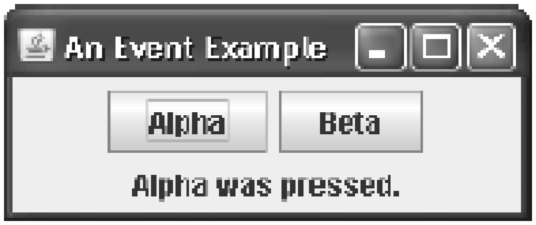
- Here, *obj is a Runnable object that will have its run( ) method called by the event dispatching* thread.
- The difference between the two methods is that invokeLater( ) returns immediately, but invokeAndWait( ) waits until obj.run( ) returns.
- You can use one of these methods to call a method that constructs the GUI for your Swing application, or whenever you need to modify the state of the GUI from code not executed by the event dispatching thread.
- You will normally want to use invokeLater( ), as the preceding program does.
- However, when constructing the initial GUI for an applet, you will need to use invokeAndWait( ).

Event handling with swings:

- Swing uses the same events as does the AWT, and these events are packaged in **java.awt.event.**
- **Events specific to Swing are stored in javax.swing.event.**

Program showing event handling with Swings:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class EventDemo {
JLabel jlab;
EventDemo() {
// Create a new JFrame container.
JFrame jfrm = new JFrame("An Event Example");
// Specify FlowLayout for the layout manager.
jfrm.setLayout(new FlowLayout());
jfrm.setSize(220, 90); // Give the frame an initial size.
// Terminate the program when the user closes the
application.
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
JButton jbtnAlpha = new JButton("Alpha"); // Button Alpha
JButton jbtnBeta = new JButton("Beta"); // Button Beta
// Add action listener for Alpha.
jbtnAlpha.addActionListener(
new ActionListener() {
public void actionPerformed(ActionEvent ae) {
jlab.setText("Alpha was pressed."); }}
);
```



Alpha was pressed.

```
// Add action listener for Beta.
jbtnBeta.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent ae) {
jlab.setText("Beta was pressed.");
}
});

// Add the buttons to the content pane.
jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);

// Create a text-based label.
jlab = new JLabel("Press a button.");

// 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 EventDemo();
}
});
}
}
```

Program explanation:

- The java.awt package is needed because it contains the FlowLayout class, which supports the standard flow layout manager used to lay out components in a frame.
- JButton provides the addActionListener( ) method, which is used to add an action listener. (JButton also provides removeActionListener( ) to remove a listener, but this method is not used by the program.

Create a Swing Applet

- JApplet is a top-level container, it includes the various panes.
- This means that all components are added to JApplet's content pane in the same way that components are added to JFrame's content pane.

  Swing applets use the same four lifecycle methods:
- init( ), start( ), stop( ), and destroy( ). Of course, you need override only those methods that are needed by your applet.

  Note:
- Painting is accomplished differently in Swing than it is in the AWT, and a Swing applet will not normally override the paint( ) method.

A simple Swing Applet Program

```
// 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().
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);
}
}
```

Swing Applet Program Explanation:

- the **init( )** method initializes the Swing components on the event dispatching thread by setting up a call to **makeGUI( ).**
- Notice that this is accomplished through the use of **invokeAndWait( )** rather than **invokeLater( )**.
- Applets must use invokeAndWait( ) because the init( ) method must not return until the entire initialization process has been completed.
- In essence, the start( ) method cannot be called until after initialization, which means that the GUI must be fully constructed.