Syllabus:

Event Handling: Two Event Handling Mechanisms, Delegation Event Model, Event Classes, Sources of Events, Listener Interfaces, Using the Delegation Event Model, Adapter Classes, Inner Classes

APPLETS: Applet Basics, Applet Architecture, An Applet Skeleton, Simple Applet Display Methods ,The AppletHTML Tag,Passing Parameters to Applets,getDocumentBase() and get Codebase(),Applet Context and show Document()

Module 4 : Event Handling

Event Handling

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events. Let's have a brief introduction to this model.

Any program that uses GUI (graphical user interface) such as Java application written for windows, is event driven. Event describes the change of state of any object.

Example: Pressing a button, Entering a character in Textbox.

Two Event Handling Mechanism

- Events are handled by the original version of java (1.0) and modern versions of Java.
- The 1.0 method of event handling is still supported, but it is not recommended for new programs.
- Many of the methods that support the old 1.0 event model have been deprecated.

1.Delegation event model :

- It defines standard and consistent mechanisms to generate and process events. Here the source generates an event and sends it to one or more listeners.
- The listener simply waits until it receives an event. Once it is obtained, It processes this event and returns.
- Listeners should register themselves with a source in order to receive an even notification. Notifications are sent only to listeners that want to receive them.

Components of Event Handling

Event handling has three main components,

Events:

- An event is a change of state of an object. 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.

Events Source:

- Event source is an object that generates an event.
- This occurs when the interval state of that object changes in some way.Source event generate more than one type of events.

Listeners:

A listener is an object that listens to the event. A listener gets notified when an event occurs.It has two major requirements

- It must have been registered with one or more source to receive notification about specific types of event
- It must implement methods to receive and process these notification

2.Event class:

The classes that represent events are at the core of Java's event handling mechanism.

Event Object: It is at the root of the Java event class hierarchy in **java.util**. It is the super class for all events.

It's one constructor is shown here: Event Object (Object *src)*

Here, *src* is the object that generates this event. Event Object contains two methods: get Source( ) and to String( ). The get Source ( ) method returns the source of the event.

Event Class	**Description**
Action Event	Generated when a button is pressed, a list item is Double-clicked, or a menu item is selected.

Adjustment Event	Generated when a scroll bar is manipulated.
Component Event	Generated when a component is hidden, moved, resized, or becomes visible.
Container Event	Generated when a component is added to or removed From a container.
Focus Event	Generated when a component gains or loses Keyboard focus.
Input Event	Abstract super class for all component input event classes.
Item Event	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.
Key Event	Generated when input is received from the keyboard.
Mouse Event	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.
Text Event	Generated when the value of a text area or text field is Changed.
Window Event	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

1.ActionEvent Class

- An **Action Event** is generated when a button is pressed, a list item is double-clicked, or
- a menu item is selected.
- The **ActionEvent** class defines four integer constants that can be used to identify any modifiers associated with an action event: **ALT_MASK,CTRL_MASK, META_MASK,** and **SHIFT_MASK.**
- **ActionEvent** has these three constructors:

  ActionEvent(Object *src*, int *type*, String *cmd)*

  ActionEvent(Object *src*, int *type*, String *cmd*, int *modifiers)*

  ActionEvent(Object *src*, int *type*, String *cmd*, long *when*, int *modifiers)*

- Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*, and its command string is *cmd*. The argument *modifiers* indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The *when* parameter specifies when the event occurred

2 The AdjustmentEvent Class

An **AdjustmentEvent** is generated by a scroll bar There are five types of adjustment events each defines integer constants that can be used to identify them

BLOCK_DECREMENT-the user clicked inside the scroll bar to decrese its value

BLOCK_INCREMENT- The user clicked inside the scroll bar to increses its value

TRACK-the slider was dragged

UNIT_DECREMENT-The button at the end of scroll bar was clicked to decreses its value

UNIT_INCREMENT-The button at the end of scroll bar was clicked to increses its value

Here is one AdjustmentEvent constructor

AdjustmentEvent(Adjustable src,int id,int type,int data);

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type and its associated data is data.*

*The **getAdjustable()** method returns the object that generated the event.*

***getAdjustmentType()** method returns one of the constant defined by the AdjustmentEvent.*

3.ComponentEvents class:

A **ComponentEvent** is generated when the size, position, or visibility of a component is changed. There are four types of component events .There are four integer constanst

COMPONENT_HIDDEN-the component was hidden

COMPONENT_MOVED-the component was moved

COMPONENT_RESIZED-the component was resized

COMPONENT_SHOWN-the component was shown.

There is one constructor

ComponentEvent(Component src,int type);

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type.*

***getComponent()** method returns the component that was generated the event.*

4.ContainerEvent class:

A **ContainerEvent** is generated when a component is added to or removed from a container

Its has two integer constant:

COMPONENT_ADDED-the component has been added to.

COMPONENT_REMOVED-The component has been removed out.

There is one constructor:

ContainerEvent(Component src,int type, component comp);

Here, *src* is a reference to the object that generated this event. The type of the

event is specified by *type and comp is the argument that indicates that component is added.*

getContainer() method generates the event.

getChild() method returns a reference to the component that was added or removed from the container.

5.ItemEvent classs:

An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected.there are two ineger constants:

DESELECTED-the user deselected an item

SELECTED-user selected an item

One constructor:

ItemEvent(itemSelectable src ,int type,object entry,int state);

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type.*

getItem()*method can be used to obtain a reference to the item that generated an event.*

getItemSelectable()*-method can be used to obtain a reference to the itemSelectable object that generated an event.*

getStateChange()*-method returns the state change for the event.*

6.KeyEvent class:

A **KeyEvent** is generated when keyboard input occurs.

There is one constructor:

KeyEvent(Component src,int type,long when,int modifier,int code,char ch);

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type.the system time at which key pressed,modifier argument indicates which modifier were pressed when key event generated.*

getChar() methods returns CHAR_UNDEFINED when a KEY_TYPED event occurs.

getKeyCode() method returns VK_UNDEFINED.

7.MouseEVENT class:

There are eight types of mouse event:

MOUSE_CLICKED-the user clicked the mouse

MOUSE_DRAGGED-the user dragged the mouse

MOUSE_ENTERED-the user entered the mouse

MOUSE_EXITED-the user exit the mouse

MOUSE_MOVED-the user moved the mouse

MOUSE_PRESSED-the user pressed the mouse

MOUSE_RELEASED-the user released the mouse

There is one constructor:

MouseEvent(Component src,int type,long when,int modifier,int x,int y,int click,boolen triggersPopup)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type.the system time at which key pressed,modifier argument indicates which modifier were pressed when key event generated.*

8.TextEvent class:

The TextEvent Class Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program.

There is one constructor:

TextEvent(Object src,int type);

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*

9.FocusEvent classs

A **Focus Event** *is generated when a component gains or losses input focus. These events*
are identified by the integer constants **FOCUS_GAINED** and **FOCUS_LOST**.
Focus Event is a subclass of **Component Event** and has these constructors:

FocusEvent(Component *src*, int *type*)
FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*)
Focus Event(Component *src*, int *type*, boolean *temporaryFlag*, Component *other*)

Here, *src* is a reference to the component that generated this event. The type of the event is specified by *type*. The argument *temporaryFlag* is set to **true** if the focus event is
temporary. Otherwise, it is set to **false**. (A temporary focus event occurs as a result of another user interface operation.

You can determine the other component by calling **getOppositeComponent( )**, shown here.

Component getOppositeComponent( )

The **isTemporary( )** method indicates if this focus change is temporary. Its form is shown here:

boolean isTemporary( )
The method returns **true** if the change is temporary. Otherwise, it returns **false**.

10.InputEvent class

The abstract class **InputEvent** is a subclass of **ComponentEvent** and is the superclass for component input events. Its subclasses are **KeyEvent** and **MouseEvent**.

InputEvent defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event. Originally, the **InputEvent** class defined the following eight values to represent the modifiers.

ALT_MASK BUTTON2_MASK META_MASK
ALT_GRAPH_MASK BUTTON3_MASK SHIFT_MASK
BUTTON1_MASK CTRL_MASK

To test if a modifier was pressed at the time an event is generated, use the **isAltDown( )**, **isAltGraphDown( )**, **isControlDown( )**, **isMetaDown( )**, and **isShiftDown( )** methods. The forms of these methods are shown here:

boolean isAltDown( )
boolean isAltGraphDown( )
boolean isControlDown( )
boolean isMetaDown( )

boolean isShiftDown( )

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

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

11.MouseWheelEvent Class

The **MouseWheelEvent** class encapsulates a mouse wheel event. It is a subclass of **MouseEvent** .

If a mouse has a wheel, it is located between the left and right buttons. Mouse wheels are used for scrolling. **MouseWheelEvent** defines these two integer constants.
WHEEL_BLOCK_SCROLL A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL A line-up or line-down scroll event occurred.

MouseWheelEvent defines the following constructor.
MouseWheelEvent(Component *src*, int *type*, long *when*, int *modifiers*,
int *x*, int *y*, int *clicks*, boolean *triggersPopup*,
int *scrollHow*, int *amount*, int *count*)

Here, *src* is a reference to the object that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when the event occurred. The coordinates of the mouse are passed in *x* and *y*. The number of clicks the wheel has rotated is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform. The *scrollHow* value must be either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_SCROLL**. The number of units to scroll is passed in *amount*. The *count* parameter indicates the number of rotational units that the wheel moved.

MouseWheelEvent defines methods that give you access to the wheel event.
To obtain the number of rotational units, call **getWheelRotation( )**, shown here.
int getWheelRotation( )

It returns the number of rotational units. If the value is positive, the wheel moved counterclockwise. If the value is negative, the wheel moved clockwise.

To obtain the type of scroll, call **getScrollType( )**, shown next.
int getScrollType( )

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

The WindowEvent Class

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

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window gained input focus.
WINDOW_ICONIFIED	The window was iconified.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.
WINDOW_STATE_CHANGED	The state of the window changed.

WindowEvent is a subclass of **ComponentEvent**. It defines several constructors. The first is
WindowEvent(Window *src*, int *type*)

Here, *src* is a reference to the object that generated this event. The type of the event is *type*.
WindowEvent(Window *src*, int *type*, int *fromState*, int *toState*)
WindowEvent(Window *src*, int *type*, Window *other*, int *fromState*, int *toState*)

Here, *other* specifies the opposite window when a focus event occurs. The *fromState*

specifies the prior state of the window and *toState* specifies the new state that the window will have when a window state change occurs.

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

3.Source Event:

Following is the list of commonly used controls while designed GUI using AWT.

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

4.Event Listeners:

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

1.ActionListener interface:

This interface define the actionPerformed() method that is invoked when an action event occurs.

void actionPerformed(ActionEvent ae);

2.AdjustmentListener interface:

This interface define the adjustmentValueChanged() method that is invoked when an adjustment event ocuurs.

void adjustmentValueChanged(AdjustmentEvent ae);

3.ComponentListener inetface:

This inetface define four methods that are invoked when a component is resized,moved,shown etc.

void componentResized(ComponentEvent ce);

void componentMoved(ComponentEvent ce);

void componentShown(ComponentEvent ce);

void componentHidden(ComponentEvent ce);

4.ItemListener inetface:

This interface define the itemStateChanged() method that is invoked when the state of an item changed.

void itemStateChanged(ItemEvent ie);

5.KeyListener interface:

This interface define three method,when key is ressed,released.

void keyPressed(KeyEvent ke);

void keyRelesed(KeyEvent ke);

void keyTyped(KeyEvent ke);

5.MouseListener interface:

This inetface define five methds

void mouseClicked(MouseEvent me);

void mouseEneterd(MouseEvent me);

void mouseExited(MouseEvent me);

void mousePressed(MouseEvent me);

void mouseReleased(MouseEvent me);

Program for handling keyboard events

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

public class Test extends Applet implements KeyListener
{
String msg="";
```

```
public void init()
{
addKeyListener(this);
}
public void keyPressed(KeyEvent k)
{
showStatus("KeyPressed");
}
public void keyReleased(KeyEvent k)
{
System.out.println("KeyRealesed");
}
public void keyTyped(KeyEvent k)
{
msg = msg+k.getKeyChar();
repaint();
}
public void paint(Graphics g)
{
g.drawString(msg, 20, 40);
}}
```

5.Adapter class:

Adapters are abstract classes for receiving various events. The methods in these classes are empty. These classes exists as convenience for creating listener objects.

AWT Adapters:

Following is the list of commonly used adapters while listening GUI events in AWT.

Sr. No.	Adapter class	Description
1	FocusAdapter	An abstract adapter class for receiving focus events.
2	KeyAdapter	An abstract adapter class for receiving key events.
3	MouseAdapter	An abstract adapter class for receiving mouse events.
4	MouseMotionAdapter	An abstract adapter class for receiving mouse motion events.
5	WindowAdapter	An abstract adapter class for receiving window events.

Program :

```
class A extends Applet
{
public void init()
{
addMoseListener(new B(this));
}
}
class B extends MouseAdater
{
A  a1;
B( A a1)
{
this.a1=a1;
}
public void mouseClicked(MouseEvent me)
{
a1.showStatus("mouse clicked");
} }
```

6.Inner class/nested class:

- Class within other class is called nested class or inner class.
- Inner class is the member of outer class
- Outer class can access all the member of inner class, where as inner class cannot access the outer class member.

Program:

```
class A extends Applet
{
public void inti()
{
```

```
addMoseListener(new B(this));
}
class B extends MouseAdater
{
public void mouseClicked(MouseEvent me)
{
a1.showStatus("mouse clicked");
}
}
}
```

APPLETS

The Applet Introduction:

- ✓ Applets are small Java program/applications that are accessed on an Internet Server, transported over the Internet, automatically installed, and run as part of a Web document

- ✓ An applet is a program written in the Java programming language that can be included in an HTML page, much in the same way an image is included in a page. When you use a Java technology enabled browser to view a page that contains an applet, the applet's code is transferred to your system and executed by the browser's Java Virtual Machine (JVM)

Two Types of Applets

There are two varieties of applets. They are

1. Based on the Applet class: **Applet**
2. Based on the Swing Class Applet: **JApplet**

1. Based on the Applet class.

- ➢ These Applet uses the Abstract Window Toolkit(AWT) to provide the graphical user interface.
- ➢ This type of applet has been widely available since java was first created.

2. Based on the Swing Class Applet.

- ➢ This applet uses the swing class to provide GUI.
- ➢ Swing offers a rich and easier to use interface than AWT.

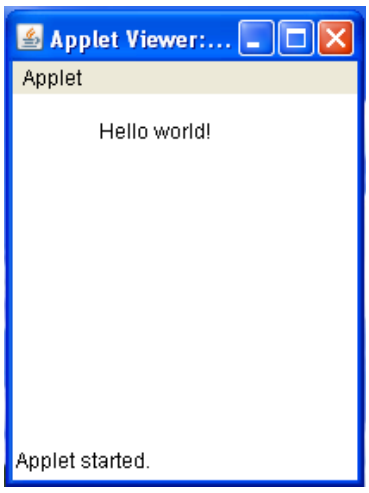
- Swing based applets are the most popular in practice.

Applet Basics

- ✓ The reason people are excited about Java as more than just another OOP language is because it allows them to write interactive applets on the web. Hello World isn't a very interactive program, but let's look at a webbed version.

```
import java.applet.Applet;
import java.awt.Graphics;


public class HelloWorldApplet extends Applet
{
     public void paint(Graphics g)
     {
          g.drawString("Hello world!", 50, 25);
     }
 }
```

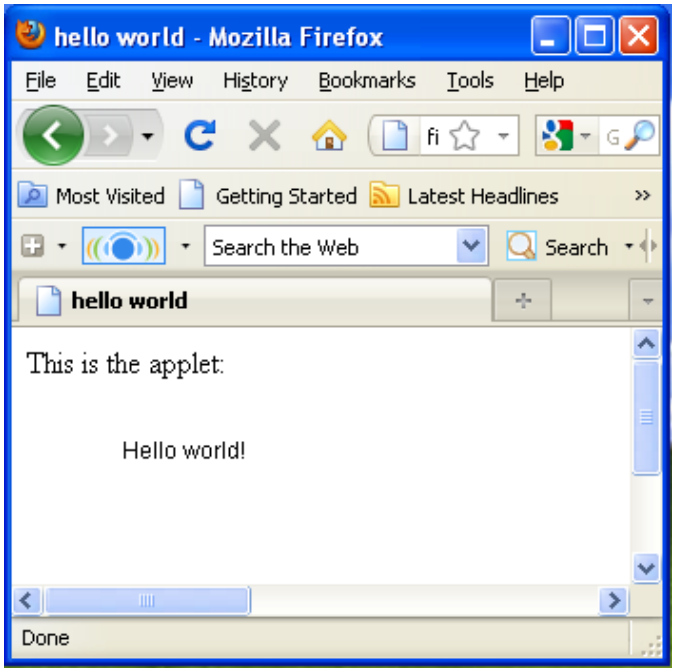
OUTPUT



- ✓ The applet version of HelloWorld is a little more complicated than the HelloWorld application, and it will take a little more effort to run it as well.
- ✓ First type in the source code and save it into file called HelloWorldApplet.java. Compile this file in the usual way. If all is well a file called HelloWorldApplet.class will be created. Now you need to create an HTML file that will include your applet. The following simple HTML file will do.

```
<html>
<head>
<title> hello world </title>
</head>

<body>
This is the applet:<P>
<applet code="HelloWorldApplet" width="150" height="50">
</applet>
</body>
</html>
```

OUTPUT



- ✓ Save this file as HelloWorldApplet.html in the same directory as the HelloWorldApplet.class file. When you have done that, load the HTML file into a Java enabled browser and see the output in the browser window.
- ✓ If the applet compiled without error and produced a HelloWorldApplet.class file, and yet you don't see the string "Hello World" in your browser chances are that the .class file is in the wrong place. Make sure HelloWorldApplet.class is in the same directory as HelloWorldApplet.html. Also make sure that your browsers support Java or that the Java plugin has been installed. Not all browsers support Java out of the box.

The Applet Class

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

Method	Summary
Void **destroy**()	Called by the browser or applet viewer to inform this applet that it is being reclaimed and that it should destroy any resources that it has allocated.
AccessibleContext **getAccessibleContext**()	Gets the AccessibleContext associated with this Applet.
AppletContext **getAppletContext**()	Determines this applet's context, which allows the applet to query and affect the environment in which it runs.
String **getAppletInfo**()	Returns information about this applet.
AudioClip **getAudioClip**(URL url)	Returns the AudioClip object specified by the URL argument.
AudioClip **getAudioClip**(URL url,	Returns the AudioClip object specified by the URL and name arguments.

String name)	
URL **getCodeBase**()	Gets the base URL.
URL **getDocumentBase**()	Returns an absolute URL naming the directory of the document in which the applet is embedded.
Image **getImage**(URL url)	Returns an Image object that can then be painted on the screen.
Image **getImage**(URL url, String name)	Returns an Image object that can then be painted on the screen.
Locale **getLocale**()	Gets the Locale for the applet, if it has been set.
String **getParameter**(String name)	Returns the value of the named parameter in the HTML tag.
String[][] **getParameterInfo**()	Returns information about the parameters than are understood by this applet.
Void **init**()	Called by the browser or applet viewer to inform this applet that it has been loaded into the system.
Boolean **isActive**()	Determines if this applet is active.
static AudioClip **newAudioClip**(URL url)	Get an audio clip from the given URL.
Void **play**(URL url)	Plays the audio clip at the specified absolute URL.
Void **play**(URL url, String name)	Plays the audio clip given the URL and a specifier that is relative to it.
Void **resize**(Dimension d)	Requests that this applet be resized.
Void **resize**(int width, int height)	Requests that this applet be resized.
Void **setStub**(AppletStub stub)	Sets this applet's stub.
Void	Requests that the argument string be displayed in the

showStatus(String msg)	"status window".
void **start**()	Called by the browser or applet viewer to inform this applet that it should start its execution.
void **stop**()	Called by the browser or applet viewer to inform this applet that it should stop its execution.

Applet Architecture

✓ An applet is a window-based program, its architecture different from the console-based programs. There are two key concepts to understand the architecture they are

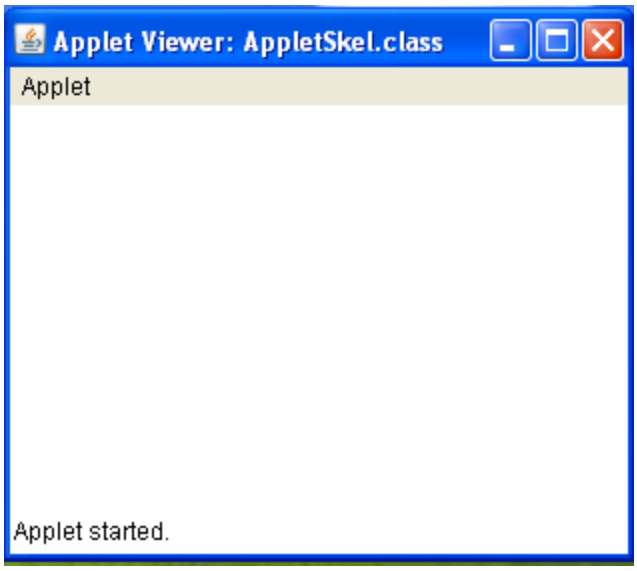
1. **Applets are Event driven**
   - An applet waits until an event occurs.
   - The *AWT* notifies the applet about an event by calling event handler that has been provided by the applet.The applet takes appropriate action and then quickly return control to *AWT*
   - All *Swing* components descend from the AWT Container class

2. **User initiates interaction with an Applet (*and not the other way aroUND*)**

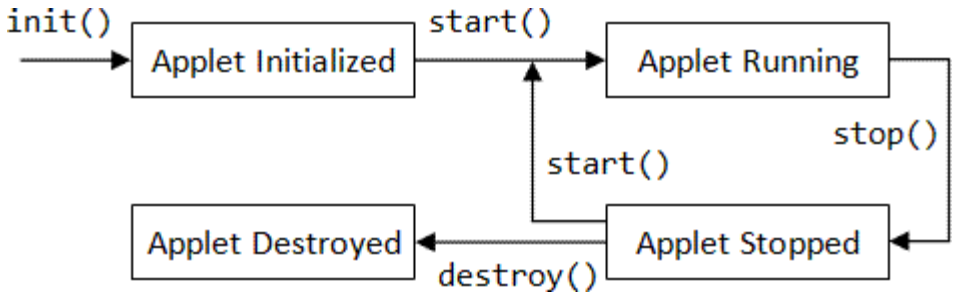
An Applet Skelton

```
// An Applet skeleton.
import java.awt.*;
import javax.swing.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/

public class AppletSkel extends JApplet
{
    // Called first.
    public void init()
    {
       // initialization
    }

  /* Called second, after init(). Also called whenever the applet is restarted. */
    public void start()
    {
        // start or resume execution
     }

  // Called when the applet is stopped.
     public void stop()
     {
           // suspends execution
      }

 /* Called when applet is terminated. This is the last method executed. */
     public void destroy()
     {
       // perform shutdown activities
     }

  // Called when an applet's window must be restored.
    public void paint(Graphics g)
    {
      // redisplay contents of window
    }
}
```

OUTPUT



Applet Initialization and Termination/Applet Life Cycle



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

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

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

1. **stop( )**
2. **destroy( )**

1. 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.

2. start( )

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

3. paint( )

The **paint( )** method is called each time your applet's output must be redrawn. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called.

The **paint( )** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

4. stop( )

The **stop( )** method is called when a web browser leaves the HTML document containing the applet when it goes to another page, for example. When **stop( )** is called, the applet is probably running. You should use **stop( )** to suspend threads

that don't need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page.

5. destroy( )

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

Simple Applet display methods

```
import java.applet.Applet;
import java.awt.Graphics;

public class HelloWorldApplet extends Applet
{
        public void paint(Graphics  g)
        {
                g.drawString("Hello world!", 50, 25);
        }
}
```

- ✓ Consider the above program to output a string to an applet, use drawString() this is a member of the Graphics class, this drawstring is called from within either update() or paint() as shown in the above program example .The general form of is

  drawString(String msg,int x, int y)

- ✓ The **msg** indicates that string to be output beginning **at x,y**. in java window the upper-left corner location is 0,0.the drawstring() method will not recognize newline character.

- ✓ To set the background color of an applet window use **setBackground**() and to set the foreground color for example the color in which text is shown use **setForeground**().these methods are defined by Component and they have the following general forms

void setBackground(Color newColor) ,

void setForeground(Color newColor)

The **newColor** specifies that new color. The class Color defines the constant shown below that can be used to specify colors.

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

Example:

setBackround(Color.cyan);

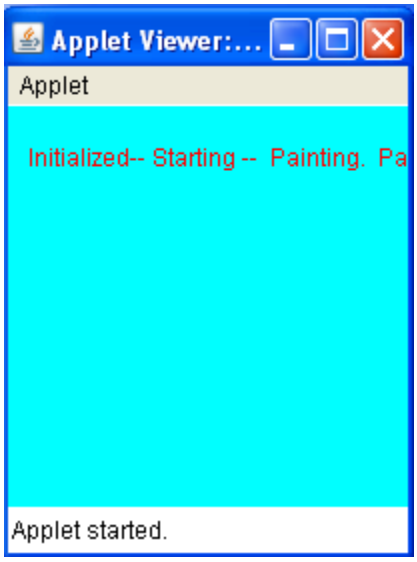
setForeground(Color.red)

Example Program:

```
/* This Applet sets the foreground and background colors and out puts a string. */
import java.applet.*;
import java.awt.*;

public class Simple extends Applet
{
        String msg;

  // set the foreground and background colors.
        public void init()
        {
          setBackground(Color.cyan);
          setForeground(Color.red);
          msg = "Initialized--";
        }
```

```
  // Add to the string to be displayed.
  public void start()
  {
        msg += " Starting --";
  }

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

OUTPUT:



Requesting repainting;

- ✓ The **repaint()** method is defined by the AWT. It causes the AWT run time system to call to your applet's update() method, which in its default implementation, calls paint(). Again for example if a part of your applet

needs to output a string, it can store this string in a variable and then call repaint(). Inside paint(), you can output the string using drawstring().

The repaint method has four forms.

```
void repaint()

void repaint(int left, int top, int width, int height)

void repaint(long maxDelay)

void repaint(long maxDelay, int x, int y, int width, int height)
```

void repaint()

This causes the entire window to be repainted

void repaint(int left, int top, int width, int height)

This specifies a region that will be repainted. the integers left, top, width and height are in pixels. You save time by specifying a region to repaint instead of the whole window.

void repaint(long maxDelay)
void repaint(long maxDelay, int x, int y, int width, int height)

Calling repaint() is essentially a request that your applet be repainted sometime soon. However, if your system is slow or busy, update() might not be called immediately. This gives rise to a problem of update() being called sporadically. If your task requires consistent update time, like in animation, then use the above two forms of repaint(). Here, the maxDelay() is the maximum number of milliseconds that can elaspe before update() is called.

Using the Status Window

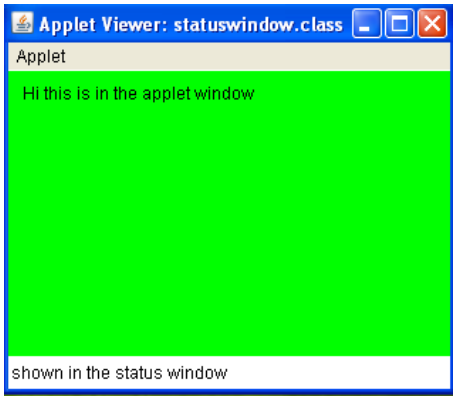
- ✓ If the user has chosen to show the Status Bar in their browser then messages can be put there from an applet.

  The **showStatus()** method would do it for this applet, if the applet was running in a browser.

Example

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

public class statuswindow extends Applet
{
        public void init()
        {
                setBackground(Color.green);

        }
        public void paint(Graphics g)
        {
        g.drawString("Hi this is in the applet window",10,20);
        showStatus("shown in the status window");
        }
}
```

OUTPUT



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. Bracketed items are optional.

```
< APPLET
      [CODEBASE = codebaseURL]
       CODE = appletFile
      [ALT = alternateText]
      [NAME = appletInstanceName]
      WIDTH = pixels HEIGHT = pixels
      [ALIGN = alignment]
      [VSPACE = pixels] [HSPACE = pixels]
>

  [< PARAM NAME = AttributeName VALUE = AttributeValue>]
  [< PARAM NAME = AttributeName2 VALUE = AttributeValue>]
  . . .
  [HTML Displayed in the absence of Java]
</APPLET>
```

CODEBASE: CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag).

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

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

WIDTH AND HEIGHT: WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.

ALIGN: ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

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

PARAM NAME AND VALUE: The PARAM tag allows you to specify appletspecific arguments in an HTML page. Applets access their attributes with the **getParameter( )** method.

Passing parameters to Applets;

- ✓ Parameters are passed to applets in **NAME=VALUE pairs in <PARAM> tags** between the opening and closing APPLET tags. Inside the applet, you read the values passed through the PARAM tags with the **getParameter() method** of the java.applet.Applet class.

The program below demonstrates this with a generic string drawing applet. The applet parameter "Message" is the string to be drawn.

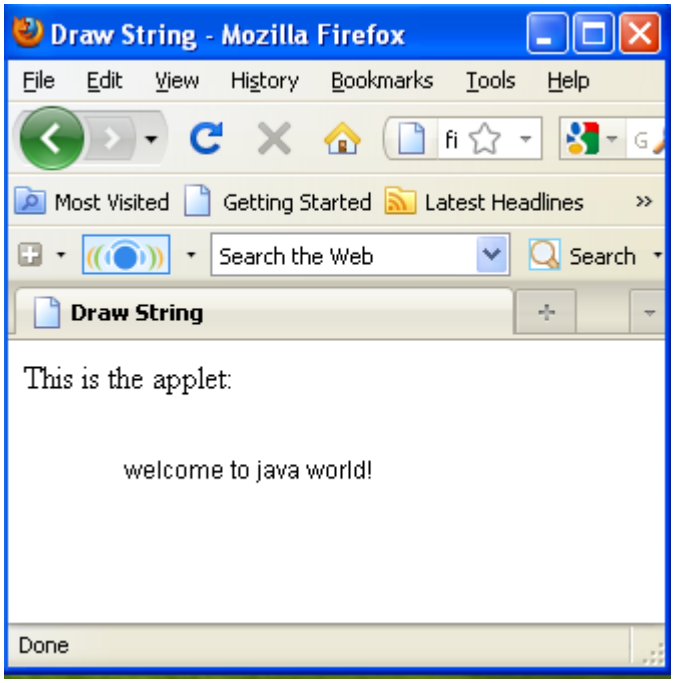
Example:

```
import java.applet.*;
import java.awt.*;
public class DrawStringApplet extends Applet
{
        private String defaultMessage = "Hello!";

        public void paint(Graphics g)
        {
              String inputFromPage = this.getParameter("Message");
               if (inputFromPage == null)

              inputFromPage = defaultMessage;

              g.drawString(inputFromPage, 50, 25);
         }
}
```

You also need an HTML file that references your applet. The following simple HTML file will do:

```
 <HTML>
<HEAD>
<TITLE> Draw String </TITLE>
</HEAD>

<BODY>
This is the applet:<P>
<APPLET code="DrawStringApplet" width="300" height="50">
<PARAM name="Message" value="welcome to java world!">
This page will be very boring if your
browser doesn't understand Java.
</APPLET>
</BODY>
</HTML>
```

OUTPUT



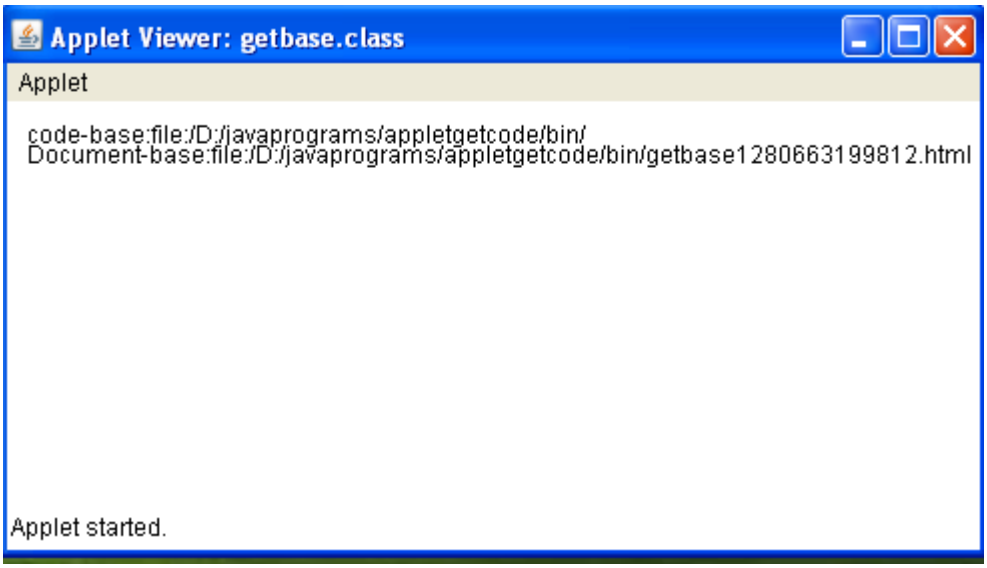
- ✓ You pass getParameter() a string that names the parameter you want. This string should match the name of a PARAM element in the HTML page. getParameter() returns the value of the parameter.
- ✓ All values are passed as strings. If you want to get another type like an integer, then you'll need to pass it as a string and convert it to the type you really want.
- ✓ The PARAM element is also straightforward. It occurs between <APPLET> and </APPLET>. It has two attributes of its own, NAME and VALUE. NAME identifies which PARAM this is. VALUE is the string value of the PARAM. Both should be enclosed in double quote marks if they contain white space.

getDocumentbase() and getCodebase()

- ✓ We sometimes need to load media and Text with the help of Applets. We have the facility to load the data from the directory which holds the HTML file which started the applet and the directory from which the applet's class loaded. These directories are returned in the form of URL by **getDocumnetBase( ) and getCodeBase( ) methods.**

```
import java.awt.*;
import java.applet.*;
import java.net.*;
public class getbase extends Applet
{
        public void paint(Graphics g)
        {
                String message;
           URL url=getCodeBase();
                message="code-base:"+url.toString();
                g.drawString(message,10,20);
                url=getDocumentBase();
                message="Document-base:"+url.toString();
                g.drawString(message,10,30);

        }
}
```

OUTPUT


Applet Viewer: getbase.class
Applet
code-base:file:/D:/javaprograms/appletgetcode/bin/
Document-base:file:/D:/javaprograms/appletgetcode/bin/getbase1280663199812.html
Applet started.

AppletContext and showDocument()

- ✓ AppletContext is an interface which helps us to get the required information from the environment in which the applet is running and getting executed.
- ✓ This information is derived by **getAppletContext( )** method which is defined by Applet. Once we get the information with the above mentioned method, we can easily bring another document into view by calling **showDocument( )** method. The basic functionality of this method is that it returns no value and never throw any exception even if it fails hence needed to be implemented with utmost care and caution.

There are two showDocument( ) methods.

1. The **method showDocument(URL)** displays the document at the specified URL.

2. The **method showDocument(URL, where)** displays the specified document at the specified location within the browser window.

```
import java.awt.*;
import java.applet.*;
import java.net.*;
public class contextdoc extends Applet
{
        public void start()
        {
                AppletContext ac=getAppletContext();
                URL url=getCodeBase();
                try
                {
                        ac.showDocument(new URL(url+"demo.html"));
                }
                catch(MalformedURLException e)
                {
                        showStatus("URL not found");

                }

        }
}
```