Java Programming – 18BCS43C

Dr. S. Chitra,
Associate Professor,
Post Graduate & Research Department of Computer Science,
Government Arts College(Autonomous), Coimbatore - 641018

Year	Subject Title	Sem.	Sub Code
2018 -19 Onwards	JAVA PROGRAMMING	IV	18BCS43C

UNIT - V

AWT and Applet: Event Handling Methods- Labels- Button Control- Check Box Control- Radio Button Control- Choice Control- List Control-Flow Layout- Border Layout-Grid Layout – Menus- Mouse Events-Applets: Life cycle of an Applet- Development and Execution of a Simple Applet.

Event and Listener (Java Event Handling)

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. The java.awt.event package provides many event classes and Listener interfaces for event handling.

Following steps are required to perform event handling:
Register the component with the Listener
Registration Methods - For registering the component with the Listener, many classes provide the registration methods. For example:

- **Button**
  - public void addActionListener(ActionListener a){}
- **MenuItem**
  - public void addActionListener(ActionListener a){}
- **TextField**
  - public void addActionListener(ActionListener a){}
  - public void addTextListener(TextListener a){}
- **TextArea**
  - public void addTextListener(TextListener a){}
- **Checkbox**
  - public void addItemListener(ItemListener a){}
- **Choice**
  - public void addItemListener(ItemListener a){}
- **List**
  - public void addActionListener(ActionListener a){}
  - public void addItemListener(ItemListener a){}

Event Class	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Java Event Handling Code

We can put the event handling code into one of the following places:

1. Within class
2. Other class
3. Anonymous class

1. Java event handling by implementing ActionListener

```
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener{
TextField tf;
AEvent(){
  //create components
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);

//register listener
b.addActionListener(this);//passing current instance

//add components and set size, layout and visibility
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
}
public static void main(String args[])
{
new AEvent();
}
}
```

public void setBounds(int xaxis, int yaxis, int width, int height); have been used in the above example that sets the position of the component it may be button, textfield etc.

2. Java event handling by outer class

```
import java.awt.*;
import java.awt.event.*;
class AEvent2 extends Frame{
TextField tf;
AEvent2(){
//create components
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);
//register listener
Outer o=new Outer(this);
b.addActionListener(o);//passing outer class instance
//add components and set size, layout and visibility
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public static void main(String args[]){
new AEvent2();
}
}
```

```
import java.awt.event.*;
class Outer implements ActionListener{
AEvent2 obj;
Outer(AEvent2 obj){
this.obj=obj;
}
public void actionPerformed(ActionEvent e){
obj.tf.setText("welcome");
}
}
```

3. Java event handling by anonymous class

```
import java.awt.*;
import java.awt.event.*;
class AEvent3 extends Frame{
TextField tf;
AEvent3(){
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(50,120,80,30);

b.addActionListener(new ActionListener(){
public void actionPerformed(){
tf.setText("hello");
}
});
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public static void main(String args[]){
new AEvent3();
}
}
```

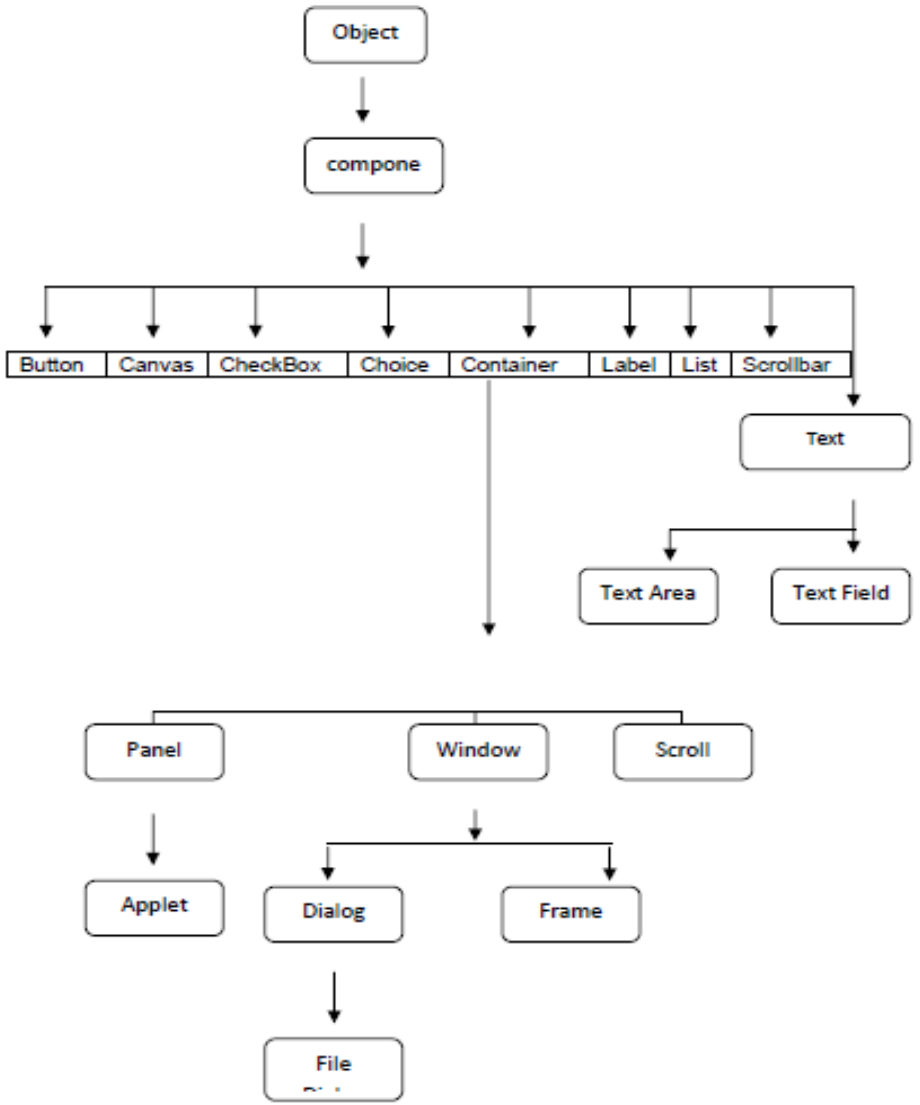
Listener Interfaces & their Description

Interfaces	Descriptions
ActionEvent	This interface is used for handling events.
Adjustable	This interface takes numeric value to adjust within the bounded range.
Composite	This interface defines methods to draw a graphical area. It combines a shape, text, or image etc.
CompositeContext	This interface allows the existence of several contexts simultaneously for a single composite object. It handles the state of the operations.
ItemSelectable	This interface is used for maintaining zero or more selection for items from the item list.

KeyEventDispatcher	The KeyEventDispatcher implements the current KeyboardFocusManager and it receives KeyEvents before dispatching their targets.
KeyEventPostProcessor	This interface also implements the current KeyboardFocusManager. The KeyboardFocusManager receives the KeyEvents after that dispatching their targets.
LayoutManager	It defines the interface class and it has layout containers.
LayoutManager2	This is the interface extends from the LayoutManager and is subinterface of that.
MenuContainer	This interface has all menu containers.

Paint	This interface is used to color pattern. It used for the Graphics2D operations.
PaintContext	This interface also used the color pattern. It provides an important color for the Graphics2D operation and uses the ColorModel.
PaintGraphics	This interface provides print a graphics context for a page.
Shape	This interface used for represent the geometric shapes.
Stroke	This interface allows the Graphics2D object and contains the shapes to outline or stylistic representation of outline.
Transparency	This interface defines the transparency mode for implementing classes.

Class hierarchy of AWT classes can be given as follows.



Labels:
This is the simplest component of Java Abstract Window Toolkit. This component is generally used to show the text or string in your application and label never perform any type of action.

Syntax for defining the label only and with justification:

Label label_name = new Label ("This is the label text.");

above code simply represents the text for the label.

Label label_name = new Label ("This is the label text. ” , Label.CENTER);

Justification of label can be left, right or centered. Above declaration used the center justification of the label using the Label.CENTER.

Example for Label.

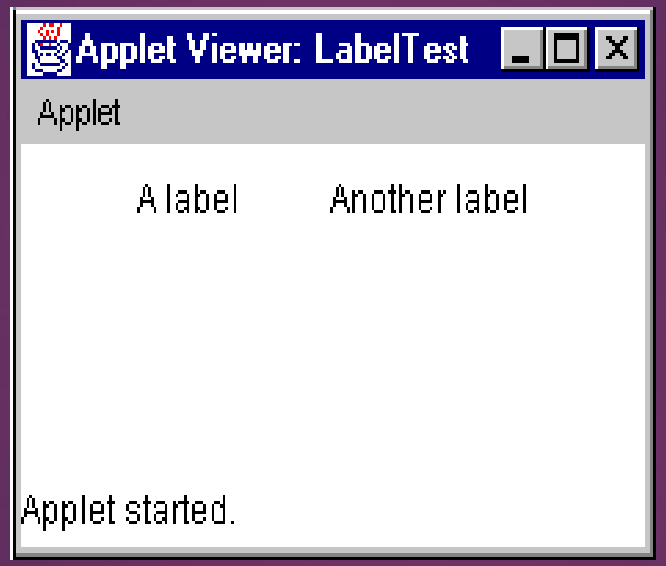
```
import java.awt.*;
import java.applet.Applet;
/*<applet code="LabelTest" width=200 height=100>
</applet>
*/
public class LabelTest extends Applet
{
public void init()
{
add(new Label("A label"));
// right justify next label
add(new Label("Another label", Label.RIGHT));
}
}
```

Save the file as **LabelTest. Java

** Compile the file using **javac LabelTest.java**

** On successful compilation, execute the file using **appletviewer LabelTest.java**

The output appers as shown in following figure :


Applet Viewer: LabelTest
Applet
A label
Another label
Applet started.

Buttons:
This is the component of Java Abstract Window Toolkit and is used to trigger actions and other events required for your application.

The syntax of defining the button is as follows:

Button button_name = new Button ("This is the label of the button.");

You can change the Button's label or get the label's text by using the

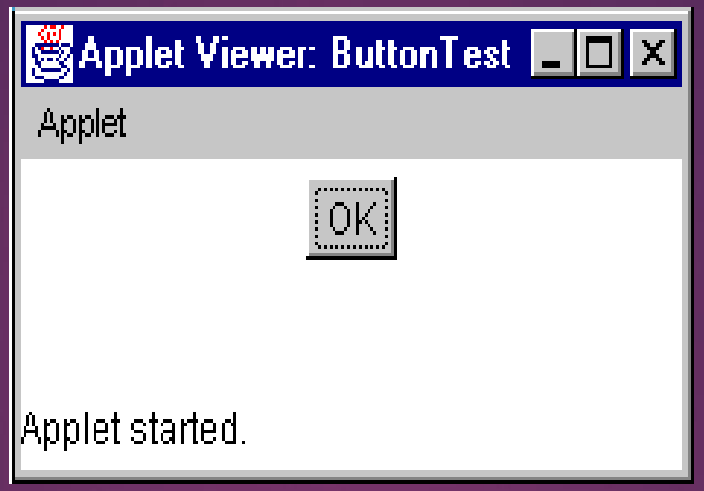
Button.setLabel (String) and Button.getLabel () method. Buttons are added to its container using the, add (button_name) method.

Example for Buttons:-

```
import java.awt.*;
import java.applet.Applet;
/*<applet code="ButtonTest" width=200 height=100>
</applet>
* /
public class ButtonTest extends Applet
{
public void init()
{
Button button = new Button ("OK");
add (button);
}
}
```

** Save the file as **ButtonTest. Java**
** Compile the file using **javac ButtonTest.java**
** On successful compilation, execute the file using **appletviewer ButtonTest.java**

The output appears as shown in following figure :



Note that in the above example there is no event handling added; pressing the button will not do anything.

Check Boxes:
This component of Java AWT allows you to create check boxes in our applications. The syntax of the definition of Checkbox is as follows:

Checkbox checkbox_name = new Checkbox ("Optional check box1", false);

Above code constructs the unchecked Checkbox by passing the boolean valued argument *false* with the Checkbox label through the Checkbox() constructor.

Defined Checkbox is added to its container using add (checkbox_name) method. You can change and get the checkbox's label using the setLabel (String) and getLabel () method.

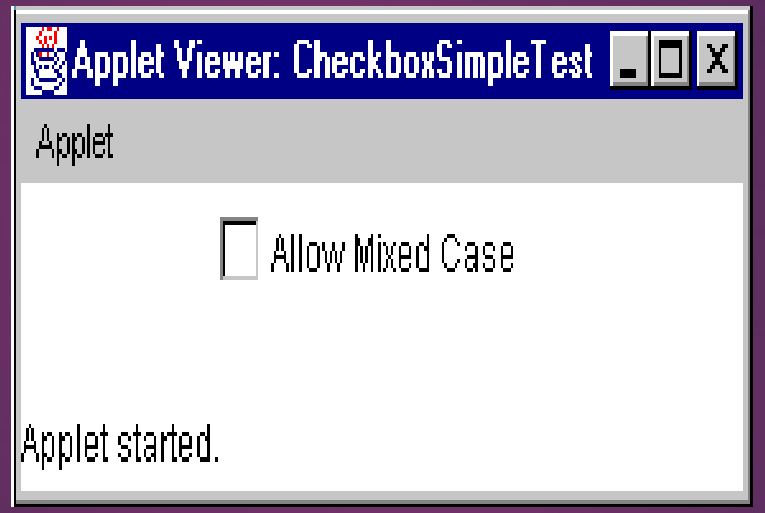
You can also set and get the state of the checkbox using the setState (boolean) and getState () method provided by the Checkbox class.

Example for Check Boxes:-

```
import java.awt.*;
import java.applet.Applet;
/*<applet code="CheckboxTest" width=200 height=100> </applet>*/
public class CheckboxTest extends Applet
{
public void init()
{
Checkbox m = new Checkbox ("Allow Mixed Case");
add (m);
}
}
```

** Save the file as **CheckboxTest. Java**
** Compile the file using **javac CheckboxTest.java**
** On successful compilation, execute the file using **appletviewer CheckboxTest.java**

The output appears as shown in following figure :



Radio Button:
Radio buttons are a bunch of option boxes in a group. Only one of then can be checked at a time. This is useful if you need to give the user a few options where only one will apply. This is the special case of the Checkbox component of Java AWT package. This is used as a group of checkboxes whos group name is same.
Only one Checkbox from a Checkbox Group can be selected at a time.

Syntax for creating radio buttons is as follows:

CheckboxGroup chkboxgp = new CheckboxGroup ();

add (new Checkbox ("chkboxname", chkboxgp, value);

“Value” in the second statement can only be true or false. If you mention more than one true valued for checkboxes then your program takes the last true and shows the last check box as checked.

Abstract Window Toolkit

AWT is a powerful concept in JAVA. AWT is basically used to develop for GUI application building. AWT is platform dependant.
That means your **.class** file after the program compilation is platform independent but the look of your GUI application is platform dependant.

AWT copies GUI component from local machines operating system. That means your applications look will differ in MAC operating system, as you have seen in WINDOWS operating system.

Some of the interfaces of AWT are

Example for Radio Buttons.

```
import java.awt.*;
import java.applet.Applet;
/*<applet code="Rbutton" width=200 height=100> </applet> */

public class Rbutton extends Applet
{
public void init()
{
CheckboxGroup chkgp = new CheckboxGroup ();
add (new Checkbox ("One", chkgp, false));
add (new Checkbox ("Two", chkgp, false));
add (new Checkbox ("Three",chkgp, false));
}
}
```

In the above code we are making three check boxes with the label "One", "Two" and "Three".

** Save the file as **Rbutton. Java**

** Compile the file using **javac Rbutton.java**

** On successful compilation, execute the file using **appletviewer Rbutton.java**

The output appears as shown in following figure :

Text Area:

This is the text container component of Java AWT package.
The Text Area contains plain text. TextArea can be declared as follows:

TextArea txtArea_name = new TextArea ();

You can make the Text Area editable or not using the setEditable (boolean) method. If you pass the boolean valued argument *false* then the text area will be non-editable otherwise it will be editable.
The text area is by default in editable mode.
Texts are set in the text area using the setText (string) method of the TextArea class.

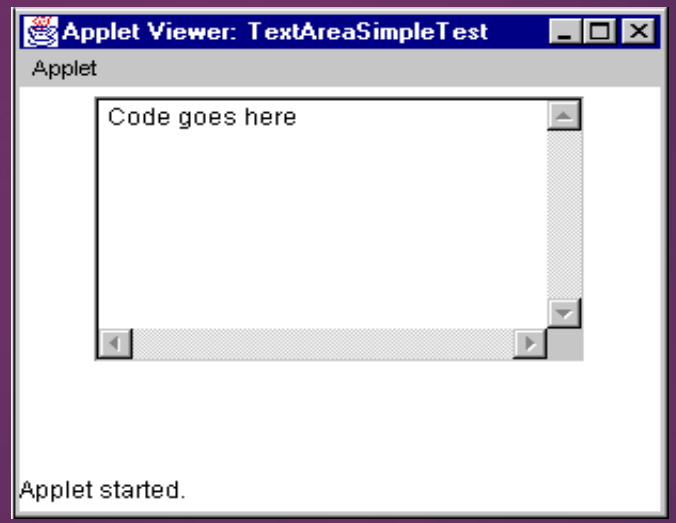
Example for Text Area:-

```
import java.awt.*;
import java.applet.Applet;
/*<applet code="TAreaTest" width=200 height=100>  </applet>*/

public class TAreaTest extends Applet
{
TextArea disp;
public void init()
{
disp = new TextArea("Code goes here", 10, 30);
add (disp);
}
}
```

** Save the file as **TAreaTest. Java**
** Compile the file using **javac TAreaTest.java**
** On successful compilation, execute the file using **appletviewer TAreaTest.java**

The output appears as shown in following figure :



Text Field:
This is also the text container component of Java AWT package.
This component contains single line and limited text information.
This is declared as follows:

TextField txtfield = new TextField (20);

You can fix the number of columns in the text field by specifying the number in the constructor.

In the above code we have fixed the number of columns to 20.

A displayed label object is known as the Label. Most of the times label is used to demonstrate the significance of the other parts of the GUI.

It helps to display the functioning of the next text field.

A label is also restricted to a single line of text as a button.

Example for Text Field:-

```
import java.awt.*;
import java.applet.Applet;
/*<applet code="TFieldTest" width=200 height=100> </applet>*/
public class TFieldTest extends Applet
{
public void init()
{
TextField f1 = new TextField("type something");
add(f1);
}
}
```

** Save the file as **TFieldTest. Java**
** Compile the file using **javac TFieldTest.java**
** On successful compilation, execute the file using **appletviewer TFieldTest.java**

The output appears as shown in following figure :

Layout Manager

The layout manager are a set of classes that implement the **java.AWT.LayoutManager** interface and help to position the components in a container.

The interface takes a task of laying out the child components in the container.

The task is achieved by resizing and moving the child components.

The advantages of this type of mechanism is that when the container is resized the layout manager automatically updates the interface

The basic layout managers includes:

1) **FlowLayout** : It is a simple layout manager that works like a word processor. It is also the default Layout manager for the panel. The flow layout lays out components linewise from left to right.

FlowLaout can be created using following constructors

a. **FlowLaout() :** Constructs a new layout with centered alignment, leaving a vertical and horizontal gap.
b. **FlowLayout(int aling, int vgap, int hgap)** : Constructs a new flowlayout with the alignment specified, leaving a vertical and horizontal gap as specified.

Various methods can be used alog with the flow layout. For eg. getAlignment(), getHgap(), getAlignment(int align) etc.

Example for Flow Layout

```
import java.awt.*;
import java.awt.event.*;
class FlowDemo extends Frame
{
Button b1 = new Button("one");
Button b2 = new Button("two");
public FlowDemo(String s)
{
super(s);
setSize(400,400);
setLayout(new FlowLayout(FlowLayout.LEFT));
addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent e)
{
System.exit(0);
}
});
add(b1);
add(b2);
}
public static void main(String arg[])
{
Frame f=new Frame();
f.show();
}
}
```

** Save the file as **FlowDemo. Java**
** Compile the file using **javac FlowDemo.java**
** On successful compilation, execute the file using **java FlowDemo.java**

2) **Grid Layout** : It lays out components in a way very similar to spread sheet (rows and columns). Specifying the number of rows and columns in grid creates the Grid layout.

Grid Layout can be created using following constructors

a. **GridLayout()** : Creates a grid layout with a default of one column per component in a single row.

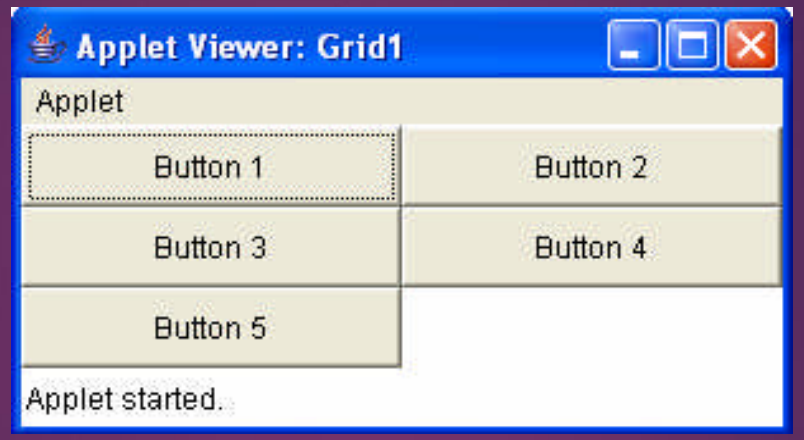
b. **GridLayout(int rows, int cols, int hgap, int vgap)** : Creates a grid layout with the specified rows and columns and specified horizontal and vertical gaps.

Various methods can be used alog with the Grid layout. For eg. getColumns(), getRows(), geHgap(), getVgap() etc.

```
import java.applet.Applet;
import java.awt.*;
public class Grid1 extends Applet {
LayoutManager Layout;
Button [ ] Buttons;
public Grid1 () {
int i;
Layout = new GridLayout (3, 2);
setLayout (Layout);
Buttons = new Button [5];
for (i = 0; i < 5; ++i) {
Buttons[i] = new Button ();
Buttons[i].setLabel ("Button " + (i + 1));
add (Buttons[i]);
}
}
}
```

** Save the file as **Grid1. Java**
** Compile the file using **javac Grid1.java**
** On successful compilation, execute the file using **appletviewer Grid1.java**

The output appears as shown in following figure :



3) **BorderLayou**t : It is the class that enables specification, i.e. where on the border of container each component should be placed. All areas need not be filled. The size of the areas will depend on the components they contain.

Border Layout can be created using following constructors

a. **BorderLayout()** : It creates a new border layout with no gap between the components.

b. **BorderLayout(int hgap, int vgap)** : It creates a border layout with the specified horizontal and vertical gap between components.

Various methods can be used alog with the Border layout. For eg. getHgap(), getVgap(), setHgap(int hgap), setVgap(int vgap)

Example for Border Layout

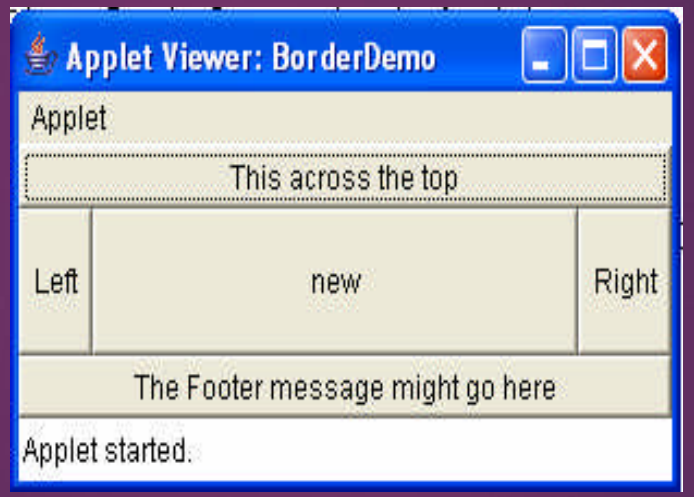
```
import java.awt.*;
import java.applet.*;
import java.util.*;
/*<applet code="BorderDemo" width=300 height=100> </applet>*/
public class BorderDemo extends Applet
{
public void init()
{
setLayout(new BorderLayout());
add(new Button("This across the top"),BorderLayout.NORTH);
add(new Button("The Footer message might go
here"),BorderLayout.SOUTH);
add(new Button("Right"),BorderLayout.EAST);
add(new Button("Left"),BorderLayout.WEST);
String msg=" This is border layout";
add(new TextArea(msg),BorderLayout.CENTER);
add(new Button("new"),BorderLayout.CENTER);
}
}
```

** Save the file as **BorderDemo. Java**

** Compile the file using **javac BorderDemo.java**

** On successful compilation, execute the file using **appletviewer BorderDemo.java**

The output appears as shown in following figure :



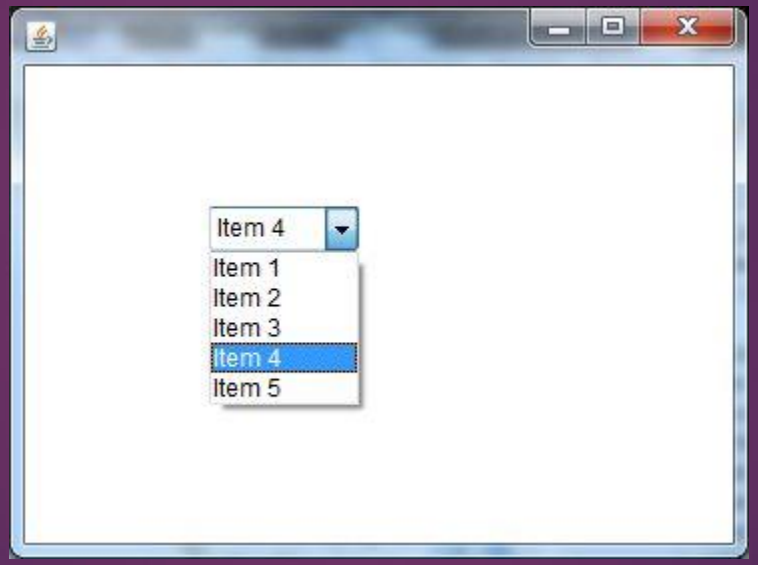
Java AWT Choice Control

The object of Choice class is used to show popup menu of choices. Choice selected by user is shown on the top of a menu. It inherits Component class.

public class Choice **extends** Component **implements** ItemSelectable, Accessible

```
import java.awt.*;
public class ChoiceExample
{
        ChoiceExample(){
        Frame f= new Frame();
        Choice c=new Choice();
        c.setBounds(100,100, 75,75);
        c.add("Item 1");
        c.add("Item 2");
        c.add("Item 3");
        c.add("Item 4");
        c.add("Item 5");
        f.add(c);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
     }
public static void main(String args[])
{
   new ChoiceExample();
}
}
```

The output appears as shown in following figure :



Java List

List in Java provides the facility to maintain the *ordered collection*.
It contains the index-based methods to insert, update, delete and search the elements.
It can have the duplicate elements also. We can also store the null elements in the list.
The List interface is found in the java.util package and inherits the Collection interface.
It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions.
The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector.

List Interface declaration

public interface List<E> **extends** Collection<E>

List Class Methods

void add(int index, E element)	It is used to insert the specified element at the specified position in a list.
boolean add(E e)	It is used to append the specified element at the end of a list.
boolean addAll(Collection<? extends E> c)	It is used to append all of the elements in the specified collection to the end of a list.
boolean addAll(int index, Collection<? extends E> c)	It is used to append all the elements in the specified collection, starting at the specified position of the list.
void clear()	It is used to remove all of the elements from this list.
boolean equals(Object o)	It is used to compare the specified object with the elements of a list.
int hashcode()	It is used to return the hash code value for a list.
E get(int index)	It is used to fetch the element from the particular position of the list.

boolean isEmpty()	It returns true if the list is empty, otherwise false.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
<T> T[] toArray(T[] a)	It is used to return an array containing all of the elements in this list in the correct order.
boolean contains(Object o)	It returns true if the list contains the specified element
boolean containsAll(Collection<?> c)	It returns true if the list contains all the specified element
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E remove(int index)	It is used to remove the element present at the specified position in the list.

int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
E remove(int index)	It is used to remove the element present at the specified position in the list.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element.
boolean removeAll(Collection<?> c)	It is used to remove all the elements from the list.
void replaceAll(UnaryOperator<E> operator)	It is used to replace all the elements from the list with the specified element.
void retainAll(Collection<?> c)	It is used to retain all the elements in the list that are present in the specified collection.
E set(int index, E element)	It is used to replace the specified element in the list, present at the specified position.
void sort(Comparator<? super E> c)	It is used to sort the elements of the list on the basis of specified comparator.

AWT MouseEvent Class

This event indicates a mouse action occurred in a component. This low-level event is generated by a component object for Mouse Events and Mouse motion events.

- a mouse button is pressed
- a mouse button is released
- a mouse button is clicked (pressed and released)
- a mouse cursor enters the unobscured part of component's geometry
- a mouse cursor exits the unobscured part of component's geometry
- a mouse is moved
- the mouse is dragged

Following is the declaration for **java.awt.event.MouseEvent** class:

```
public class MouseEvent extends InputEvent
```

Field
Following are the fields for **java.awt.event.MouseEvent** class:

- **static int BUTTON1** --Indicates mouse button #1; used by getButton()
- **static int BUTTON2** --Indicates mouse button #2; used by getButton()
- **static int BUTTON3** --Indicates mouse button #3; used by getButton()
- **static int MOUSE_CLICKED** --The "mouse clicked" event
- **static int MOUSE_DRAGGED** --The "mouse dragged" event
- **static int MOUSE_ENTERED** --The "mouse entered" event
- **static int MOUSE_EXITED** --The "mouse exited" event
- **static int MOUSE_FIRST** --The first number in the range of ids used for mouse events
- **static int MOUSE_LAST** -- The last number in the range of ids used for mouse events
- **static int MOUSE_MOVED** --The "mouse moved" event
- **static int MOUSE_PRESSED** -- The "mouse pressed" event
- **static int MOUSE_RELEASED** --The "mouse released" event
- **static int MOUSE_WHEEL** --The "mouse wheel" event
- **static int NOBUTTON** --Indicates no mouse buttons; used by getButton()
- **static int VK_WINDOWS** --Constant for the Microsoft Windows "Windows" key.

MouseEvent Class Methods

1	**int getButton()** Returns which, if any, of the mouse buttons has changed state.
2	**int getClickCount()** Returns the number of mouse clicks associated with this event.
3	**Point getLocationOnScreen()** Returns the absolute x, y position of the event.
4	**static String getMouseModifiersText(int modifiers)** Returns a String describing the modifier keys and mouse buttons that were down during the event, such as "Shift", or "Ctrl+Shift".
5	**Point getPoint()** Returns the x,y position of the event relative to the source component.
6	**int getX()** Returns the horizontal x position of the event relative to the source component.
7	**int getXOnScreen()** Returns the absolute horizontal x position of the event.
8	**int getY()** Returns the vertical y position of the event relative to the source component.
9	**int getYOnScreen()** Returns the absolute vertical y position of the event.
10	**boolean isPopupTrigger() Returns whether or not this mouse event is the popup menu trigger event for the platform.**
11	**String paramString()** Returns a parameter string identifying this event.
12	**void translatePoint(int x, int y)** Translates the event's coordinates to a new position by adding specified x (horizontal) and y (vertical) offsets.

Java AWT MenuItem and Menu

The object of MenuItem class adds a simple labeled menu item on menu. The items used in a menu must belong to the MenuItem or any of its subclass.

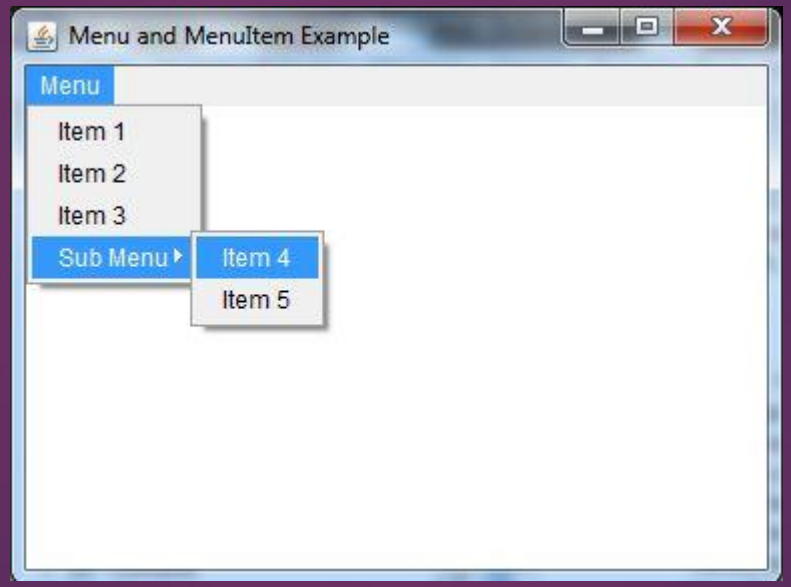
The object of Menu class is a pull down menu component which is displayed on the menu bar. It inherits the MenuItem class.

AWT MenuItem class declaration
public class MenuItem **extends** MenuComponent **implements** Accessible

AWT Menu class declaration
public class Menu **extends** MenuItem **implements** MenuContainer, Accessible

```
import java.awt.*;
class MenuExample
{
    MenuExample(){
        Frame f= new Frame("Menu and MenuItem Example");
        MenuBar mb=new MenuBar();
        Menu menu=new Menu("Menu");
        Menu submenu=new Menu("Sub Menu");
        MenuItem i1=new MenuItem("Item 1");
        MenuItem i2=new MenuItem("Item 2");
        MenuItem i3=new MenuItem("Item 3");
        MenuItem i4=new MenuItem("Item 4");
        MenuItem i5=new MenuItem("Item 5");
        menu.add(i1);
        menu.add(i2);
        menu.add(i3);
        submenu.add(i4);
        submenu.add(i5);
        menu.add(submenu);
        mb.add(menu);
        f.setMenuBar(mb);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
}
public static void main(String args[])
{
new MenuExample();
}
}
```

The output appears as shown in following figure :



Applet Programming

There are two kinds of Java programs, applications (also called stand-alone programs) and Applets. An **Applet** is a small Internet-based program that has the Graphical User Interface (GUI), written in the Java programming language.

Applets are small Java programs that are primarily used in Internet computing. They can be transported over the Internet from one computer to another and run using the **Applet Viewer** or any Web browser that supports Java. An applet, like any application program, can do many things for us. It can perform arithmetic operations, display graphics, play sounds, accept user input, create animation, and play interactive games.

Java has revolutionized the way the Internet users retrieve and use documents on the world wide network. Java has enabled them to create and use fully interactive multimedia Web documents. A web page can now contain not only a simple text or a static image but also a Java applet which, when run, can produce graphics, sounds and moving images. Java applets therefore have begun to make a significant impact on the World Wide Web.

Applets are designed to run inside a web browser or in applet viewer to facilitate the user to animate the graphics, play sound, and design the GUI components such as text box, button, and radio button. When applet arrives on the client, it has limited access to resources, so that it can produce arbitrary multimedia user interface and run complex computation without introducing the risk of viruses or breaching data integrity.

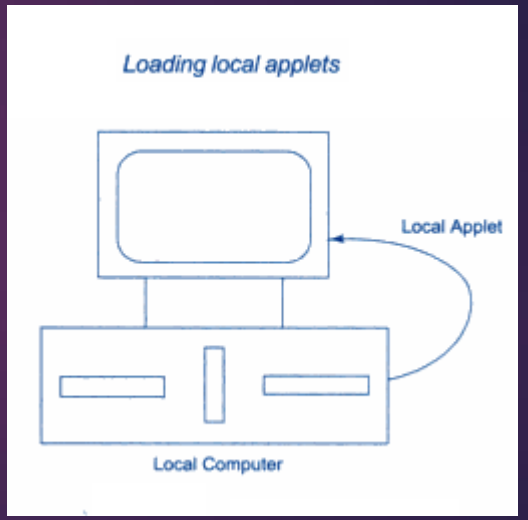
To create an applet, we extend the “java.applet.Applet” class and by overriding the methods of java.awt.Applet, new functionality can be placed into web pages.

Applets are compiled using javac compiler and it can be executed by using an appletviewer or by embedding the class file in the HTML (Hyper Text Markup Languege) file.

Local and Remote Applets

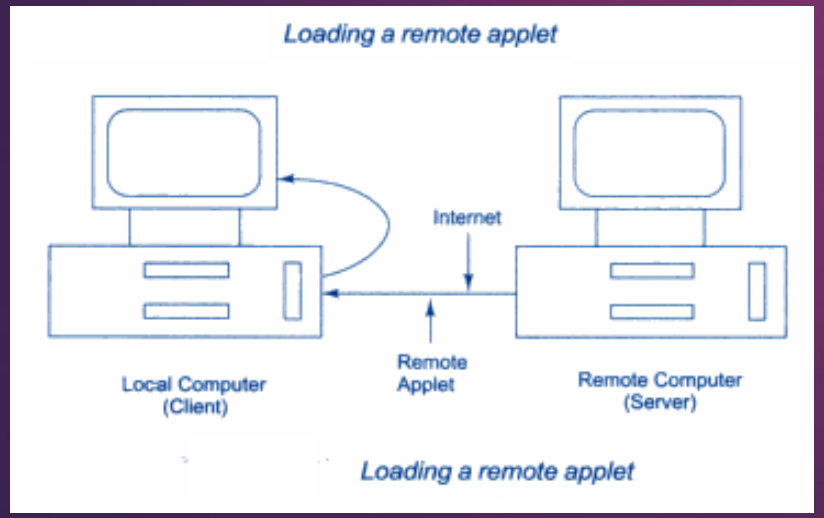
We can embed applets into Web pages in two ways. One, we can write our own applets and embed them into Web pages. Second, we can download an applet from a remote computer system and then embed it into a Web page.

An applet developed locally and stored in a local system is known as a *local applet*. When a Web page is trying to find a local applet, it does not need to use the Internet and therefore the local system does not require the Internet connection. It simply searches the directories in the local system and locates and loads the specified applet



A *remote applet* is that which is developed by someone else and stored on a remote computer connected to the Internet. If our system is connected to the Internet, we can download the remote applet onto our system via at the Internet and run it

In order to locate and load a remote applet, we must know the applet's address on the Web. This address is known as *Uniform Resource Locator (URL)* and must be specified in the applet's HTML document as the value of the CODEBASE attribute




```
CODEBASE = http : // www.netserve.com / applets
```

In the case of local applets, CODEBASE may be absent or may specify a local directory.

How Applets Differ from Applications

- Applets do not use the **main( )** method for initiating the execution of the code. Applets, when loaded, automatically call certain methods of Applet class to start and execute the applet code.
- Unlike stand-alone applications, applets cannot be run independently. They are run from inside a Web page using a special feature known as HTML tag.
- Applets cannot read from or write to the files in the local computer.
- Applets cannot communicate with other servers on the network.
- Applets cannot run any program from the local computer.
- Applets are restricted from using libraries from other languages such as C or C++. (Remember, Java language supports this feature through **native** methods).

All these restrictions and limitations are placed in the interest of security of systems. These restrictions ensure that an applet cannot do any damage to the local system.

* Applets are designed just for handling the client site problems. while the java applications are designed to work with the client as well as server.

Preparing to Write Applets

First of all, let us consider the situations when we might need to use applets.

1. When we need something dynamic to be included in the display of a Web page. For example, an applet that displays daily sensitivity index would be useful on a page that lists share prices of various companies or an applet that displays a bar chart would add value to a page that contains data tables.
2. When we require some "flash" outputs. For example, applets that produce sounds, animations or some special effects would be useful when displaying certain pages.
3. When we want to create a program and make it available on the Internet for us by others on their computers.

Before we try to write applets, we must make sure that Java is installed properly and also ensure that either the Java **appletviewer** or a Java-enabled browser is available. The steps involved in developing and testing in applet are:

1. Building an applet code (**.java** file)
2. Creating an executable applet (**.class** file)
3. Designing a Web page using HTML tags
4. Preparing <APPLET> tag
5. Incorporating <APPLET> tag into the Web page
6. Creating HTML file
7. Testing the applet code

1. Building Applet Code (.java file)

It is essential that our applet code uses the services of two classes, namely, **Applet** and **Graphics** from the Java class library. The **Applet** class which is contained in the **java.applet** package provides life and behaviour to the applet through its methods such as **init( )**, **start( )** and **point( )**. Unlike the applications, where Java calls the **main( )** method directly to initiate the execution of the program, when an applet is loaded, Java automatically calls a series of **Applet** class methods for starting, running, and stopping the applet code. The **Applet** class therefore maintains the *lifecycle* of an applet.

The **paint( )** method of the **Applet** class, when it is called, actually displays the result of the applet code on the screen. The output may be text, graphics, or sound. The **paint( )** method, which requires a **Graphics** object as an argument, is defined as follows:

```
public void paint (Graphics g)
```

This requires that the applet code imports the **java.awt** package that contains the **Graphics** class. All output operations of an applet are performed using the methods defined in the **Graphics** class. It is thus clear from the above discussions that an applet code will have a general format as shown below:

```
import  java.awt.*;
import  java.applet.*;
..........
..........
public  class  appletclassname  extends  Applet
{
     ..........
     ..........
     public  void  paint  (Graphics  g)
     {
          ..........
          ..........        / /  Applet operations code
          ..........
     }
     ..........
     ..........
}
```

The *appletclassname* is the main class for the applet. When the applet is loaded, Java creates an instance of this class, and then a series of **Applet** class methods are called on that instance to execute the code.

APPLET CLASS

The **java.applet** package is the smallest package in Java API(Application Programming Interface). The **Applet class** is the only class in the package. The Applet class has many methods that are used to display images, play audio files etc but it has no main() method. Some of them were explained below that give you the knowledge about Applets and their behavior.

init() : This method is used for whatever initializations are needed for your applet. Applets can have a default constructor, but it is better to perform all initializations in the init method instead of the default constructor.

start() :This method is automatically called after Java calls the init method. If this method is overwritten, code that needs to be executed every time that the user visits the browser page that contains this applet.

stop() : This method is automatically called when the user moves off the page where the applet sits. If your applet doesn't perform animation, play audio files, or perform calculations in a thread, you don't usually need to use this method.

destroy(): Java calls this method when the browser shuts down.

Advantages of a Java Applet:

Following are the advantages of a Java Applet:

1. The most Important feature of an Applet is, It is truly platform independent so there is no need of making any changes in the code for different platform i.e. it is simple to make it work on Linux, Windows and Mac OS i.e. to make it cross platform.
2. The same applet can work on "all" installed versions of Java at the same time, rather than just the latest plug-in version only.
3. It can move the work from the server to the client, making a web solution more scalable with the number of users/clients.
4, The applet naturally supports the changing user state like figure positions on the chessboard.
5. Applets improves with use: after a first applet is run, the JVM is already running and starts quickly.
6. Applets can be used to provide dynamic user-interfaces and a variety of graphical effects for web pages.

Every java Applet inherits a set of default behaviours from the Applet class. As a result, when an applet is loaded it undergoes a series of changes in its state. Following are the states in applets lifecycle.

1) **Born or Initialisation state:**
An applet begins its life when the web browser loads its classes and calls its init() method. This method is called exactly once in Applets lifecycle and is used to read applet parameters. Thus, in the init() method one should provide initialization code such as the initialization of variables.
Eg. public void init()
{
//initialisation
}

2) **Running State:**
Once the initialization is complete, the web browser will call the start() method in the applet. This method must called atleast once in the Applets lifecycle as the start() method can also be called if the Applet is in “Stoped” state. At this point the user can begin interacting with the applet.

Eg. public void start()
{
//Code
}

3) **Stopped State:**
The web browser will call the Applets stop() method, if the user moved to another web page while the applet was executing. So that the applet can take a breather while the user goes off and explores the web some more. The stop() method is called atleast once in Applets Lifecycle.

Eg. publc void stop()
{
//Code
}

4) **Dead State:**

Finally, if the user decides to quit the web browser, the web browser will free up system resources by killing the applet before it closes. To do so, it will call the applets destroy() method. One can override destroy() to perform one-time tasks upon program completion. for example, cleaning up threads which were started in the init() method.

Eg. public void destroy()
{
// Code
}

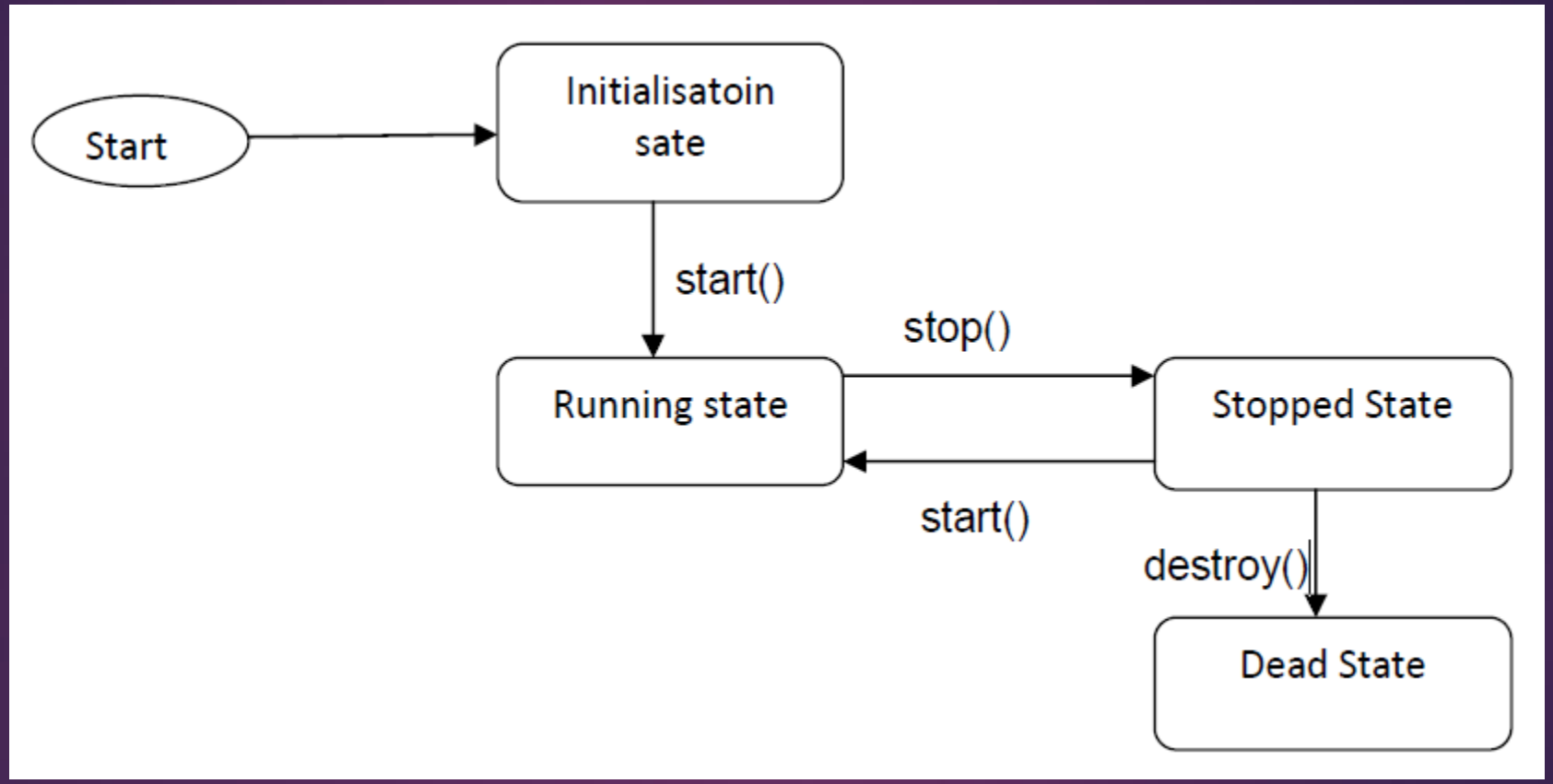
Note: If the user returns to the applet, the web browser will simply call the applet's start() method again and the user will be back into
the program

5) **Display State :**

Applet moves to the display state whenever it has to perform the output operations on the screen. This happens immediately after the applet enters into the running state. The paint() method is called to accomplish this task.

Eg. public void paint(Graphics g)
{
//Display Statements
}

Lifecycle of an Applet Graphically



The following example is made simple enough to illustrate the essential use of Java applets through its java.applet package.

Example.

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet
{
public void paint(Graphics g)
{
g.drawString("My First Applet",40,40);
}
}
```

** Save the file as **SimpleApplet.java**

** Compile the file using **javac SimpleApplet.java**

Here is the illustration of the above example,

1. In the first line we imorts the Abstract Window Toolkit(AWT) classes as Applet interact with the user through the AWT, not through the console – based I/O classes. The AWT contains support for a window based graphical interface.

2. In the second line we import the Applet package, which contains the class “Applet”. As every applet that we create is the subclass of Applet.

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

4. Inside simpleApplet, paint() method is declared. This method is defined by the AWT and must be overridden by the Applet. Method paint() is called each time that the applet must redisplay its output.

5. This paint() method has parameter of type “Graphics”. This parameter contains the graphics context, which describes the graphics environment in which the
applet is running. This context is used whenever output to the applet is required.

6. Inside paint() method is a call to drawstring(), which is a member of the Graphics class. This method output a String beginning at specified X, Y locations.

2. Creating an Executable file

Executable applet is nothing but the **.class** file of the applet, which is obtained by compiling the source code of the applet. Compiling an applet is exactly the same as compiling an application. Therefore, we can use the Java compiler to compile the applet.

Let us consider the **HelloJava** applet created This applet has been stored in a file called **HelloJava.java**. Here are the steps required for compiling the **HelloJava** applet.

1. Move to the directory containing the source code and type the following command:

```
javac HelloJava.java
```

2. The compiled output file called **HelloJava.class** is placed in the same directory as the source.
3. If any error message is received, then we must check for errors, correct them and compile the applet again.

3. Designing a Webpage

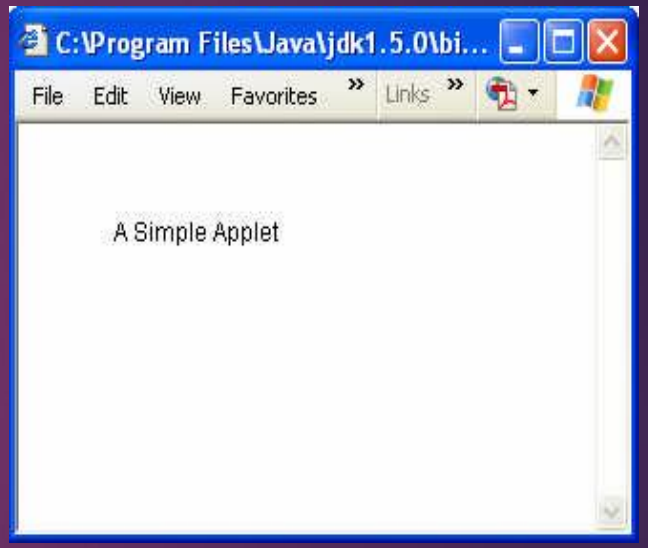
There are two ways in which one can run an applet, as
follows
1) Executing the applet within a java-compatible web browser.
2) Using an applet viewer, such as the standard SDK tool, “appletviewer”. An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.
**To execute an applet in a web browser, you need to write a short HTML text file that contains the appropriate APPLET tag.
For above example it is

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

1. Save this code in text file with extension **.html** say **Myapplet.html.**
2. Compile the file using **javac SimpleApplet.java**
3. On successful compilation of SimpleApplet.java file, execute the this file using a**ppletviewer Myapplet.html** or just open this html file dirctly.

The output of above example appears as shown in the following figure:

OR



4. Preparing Applet Tag

The Applet tag is used to start an applet from both HTML document and form 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 in a single
page.
The <applet....> tag included in the body section of HTML file supplies the name of the applet to be loaded and tells the browser how much space the applet requires

The syntax for the standard Applet tag is as follows

<applet[codebase=codebaseURL] code=”Applet file”
[ALT=”alternative text]
[name=AppletInstanceName]
Width=pixels height= pixels
[align= alignment]
>
[<param name=”Attributename” value =”Attribute value”]
[<param name=”Attributename” value =”Attribute value”]
.....
[HTML displayed in the absence of java]
</applet>

Here is meaning of each piece of above code

1. **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. The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read.

2. **Code**: code is required attribute that gives the name of the file containing the applets compiled .class file. This file is relative t the code base URL of the applet , which is the directory that the HTML file was in or the directory indicated by the CODEBASE if set.

3. ALT : The ALT tag is an optional attribute used to specify a short text message that should be displayed if browser understand the APPLET tag but cant currently run java applet.

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

5. **Param name and value** : The PARAM tag allows us to specify applet specific arguments in an HTML page. Applets access their attributes with the getParameter() method.

5. Incorporating the Applet Tag into Web page

One can supply user-defined parameters to an applet using <param.....> tag. Each <param....> tag has a **name** attribute such as color,and a **value** attribute such as red. Inside the applet code, the applet can refer to that parameter by name to find its value. For e.g. the color of the text can be changed to red by an applet using a
<param...> tag as follows

<applet....>
<param=color value = “red”>
</applet>
Similarly we can change the text to be displayed by an
applet by supplying new text to the applet through a <param....>tag
as shown below.
<param name=text value = “xyz” >

Passing a parameters to an applet is similar to passing parameters to main() method using command line arguments. To set up and handle parameters, we need to do two things.

1) Include appropriate <param.....> tags in the HTML document.

2) Provide code in the applet to pass these paraments. Parameters are passed to an applet when it is loaded. We can define the init() method in the applet to get hold of the parameters defined in the <param> tags. This is done using the getparameter() method, which takes one string argument representing the name of the parameter and returns a string containing the value of that parameter.

6. Aligning the Applet

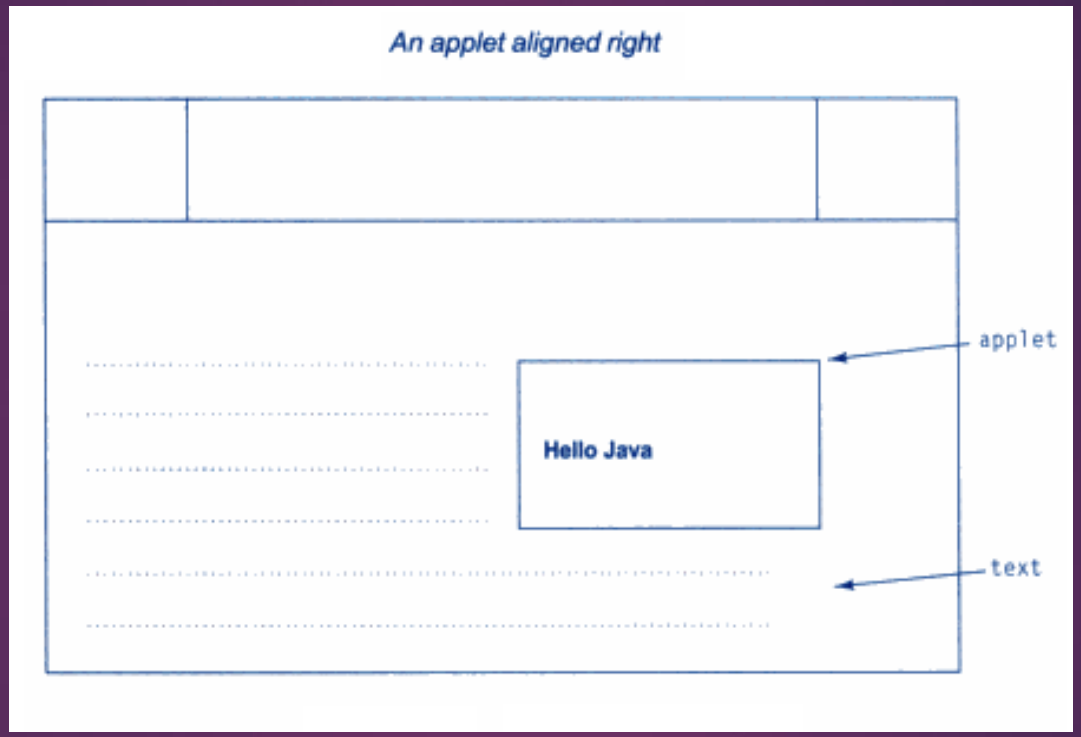
We can align the output of the applet using the ALIGN attribute. This attribute can have one of the nine values:

LEFT, RIGHT, TOP, TEXT TOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM, ABSBOTTOM.

For example, ALGN = LEFT will display the output at the left margin of the page. All text that follows the ALIGN in the Web page will be placed to the right of the display. Program shows a HTML file for our **HelloJava** applet shown in Program .

Program ***HTML file with ALIGN attribute***

```
<HTML>
      <HEAD>
      <TILTE> Here is an applet </TITLE>
      </HEAD>
      <BODY>
            <APPLET CODE   = HelloJava.class
                    WIDTH  = 400
                    HEIGHT = 200
                    ALIGN  = RIGHT >
            </APPLET>
      </BODY>
</HTML>
```


An applet aligned right
Hello Java
applet
text

Running the Applet

Now that we have created applet files as well as the HTML file containing the applet, we must have the following files in our current directory:

```
HelloJava.java
HelloJava.class
HelloJava.html
```

To run an applet, we require one of the following tools:

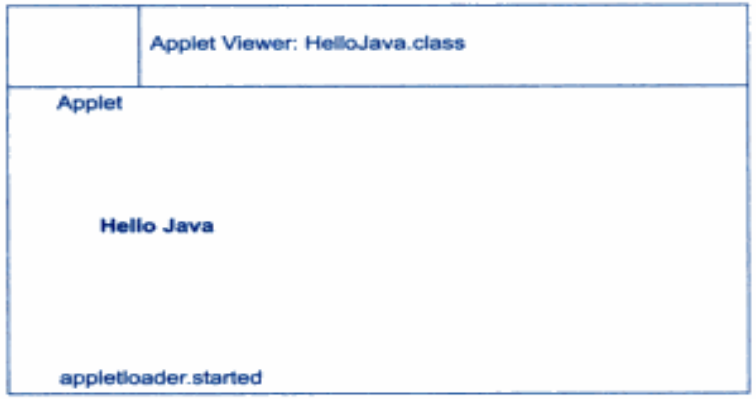
1. Java-enabled Web browser (such as HotJava or Netscape)
2. Java appletviewer

If we use a Java-enabled Web browser, we will be able to see the entire Web page containing the applet. If we use the **appletviewer** tool, we will only see the applet output. Remember that the **appletviewer** is not a full-fledged Web browser and therefore it ignores all of the HTML tags except the part pertaining to the running of the applet.

The **appletviewer** is available as a part of the Java Development Kit that we have been using so far. We can use it to run our applet as follows:

```
appletviewer HelloJava.html
```

Notice that the argument of the **appletviewer** is not the **.java** file or the **.class** file, but rather **.html** file. The output of our applet will be as shown in Fig.



Output of HelloJava applet by using appletviewer

The list of things to be done for adding an applet to a HTML document:

1. Insert an <APPLET> tag at an appropriate place in the Web page.
2. Specify the name of the applet's **.class** file.
3. If the **.class** file is not in the current directory, use the codebase parameter to specify
   - the relative path if file is on the local system, or
   - the Uniform Resource Locator (URL) of the directory containing the file if it is on a remote computer.
4. Specify the space required for display of the applet in terms of width and height in pixels.
5. Add any user-defined parameters using <PARAM> tags.
6. Add alternate HTML text to be displayed when a non-Java browser is used.
7. Close the applet declaration with the </APPLET> tag.

References:

1. “Programming With JAVA – A Primer” , E. Balagurusamy, Fourth Edition, TMH Publications

2. “Java Programming” Internet Content – Anonomous Author

3. “The complete reference JAVA2”, Hervert schildt. TMH Publications.