Unit – 5

Introduction to Applet

Applets are small Java programs that are used in Internet computing. The Applets can be easily transported over the internet from one computer to another computer and can be run using "**appletviewer**" or java enabled "**Web Browser**". An Applet like any application program can do many things for us. It can perform arithmetic operations, display graphics, animations, text, accept the user input and play interactive games.

Applets are created in the following situations:

1. When we need something to be included dynamically in the web page.
2. When we require some flash output
3. When we want to create a program and make it available on the internet.

Applet Basics

- All applets are subclasses (either directly or indirectly) of **Applet**. Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer, which is provided by the JDK.
- Execution of an applet does not begin at **main( )**.
- Output to your applet's window is not performed by **System.out.println( )**. Rather, in non-Swing applets, output is handled with various AWT methods, such as **drawString( )**, which outputs a string to a specified X,Y location
- To use an applet, it is specified in an HTMLfile. One way to do this is by using the APPLET tag. **(HTML stands for Hyper Text Markup Language)**
- The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTMLfile.
- To just test the applet it can be executed using the appletviewer. The applet tag must be included as comment lines in the java source program as follow: **/*<applet code="applet_name" width=400 height=400 ></applet> */**
- To view the applet using the HTML file, it can be included in the HTML file with <applet>.
- Tag as follow:

Filename.HTML

```
<html>
<head><title> The name of the Web Page</title>
</head>
<body>
<applet code="applet_name"   width=400 height=400 ></applet>
</body>
</html>
```

Note: Here, the **<applet>** is the name of the tag, and "code" ,"width" and "height" are called attributes of the Tag, applet_name, 400, 400 are called values respectively.

Applet Architecture

An applet is a window-based program. As such, its architecture is different from the console-based programs. The key concepts are as follow:

First, applets are event driven. An applet waits until an event occurs. The run-time system notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return.

Second, the user initiates interaction with an applet. These interactions are sent to the applet as events to which the applet must respond. For example, when the user clicks the mouse inside the applet's window, a mouse-clicked event is generated. If the user presses a key while the applet's window has input focus, a keypress event is generated

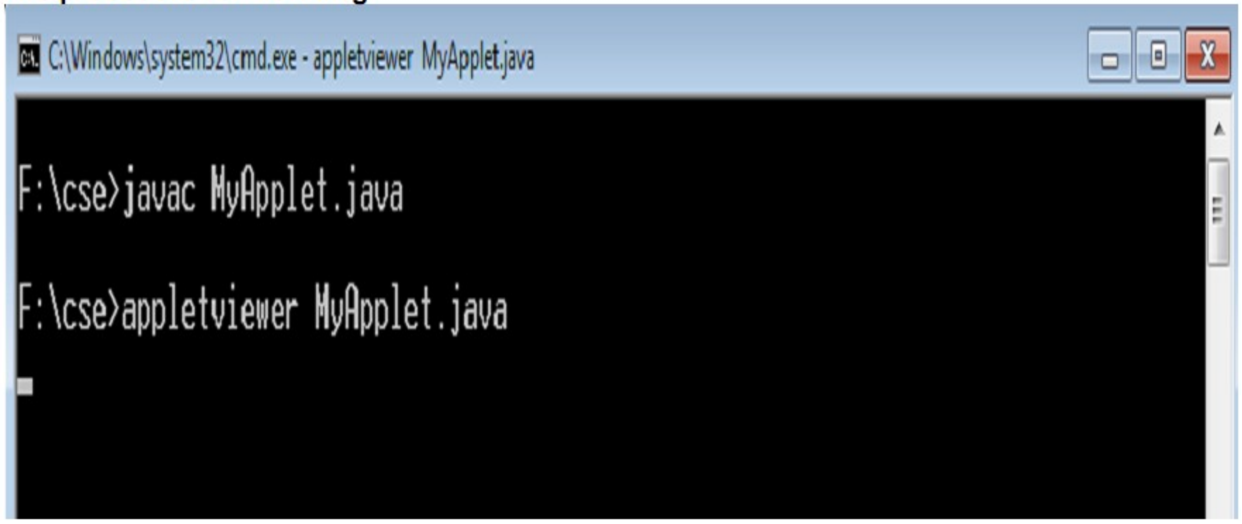
An Applet Skelton – An Example of Applet

Most of the applets override a set of methods of the Applet. Four of these methods, **init( )**, **start( )**, **stop( )**, and **destroy( )**, apply to all applets and are defined by **Applet**. Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use.

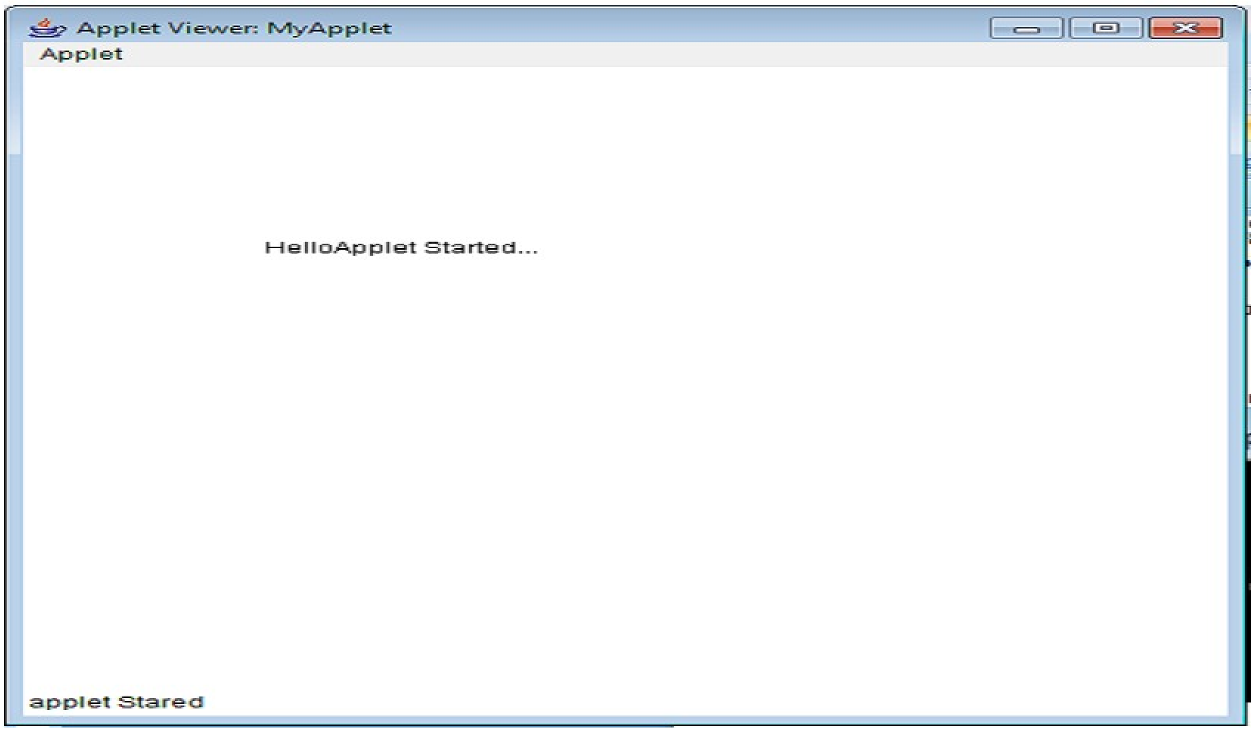
AWT-based applets will also override the **paint( )** method, which is defined by the AWT **Component** class. This method is called when the applet's output must be redisplayed. These five methods can be assembled into the skeleton shown here:

```
// An Applet skeleton. import java.awt.*;
import java.applet.*;
/* <applet code="AppletSkel" width=300 height=100> </applet> */
public class AppletSkel extends Applet
{ // Called first.
public void init()
{  // initialization
}

/* Called second, after init(). Also called whenever the applet is restarted. */

public void start()
{
// start or resume execution
}
// Called when the applet is stopped.
public void stop()
{
// suspends execution
}
 /* Called when applet is terminated. This is the last  method executed. */
public void destroy()
{  // perform shutdown activities
}
// Called when an applet's window must be restored. public void paint(Graphics g)
{  // redisplay contents of window
} }
```

The procedure for Running



```
F:\cse>javac MyApplet.java

F:\cse>appletviewer MyApplet.java
```

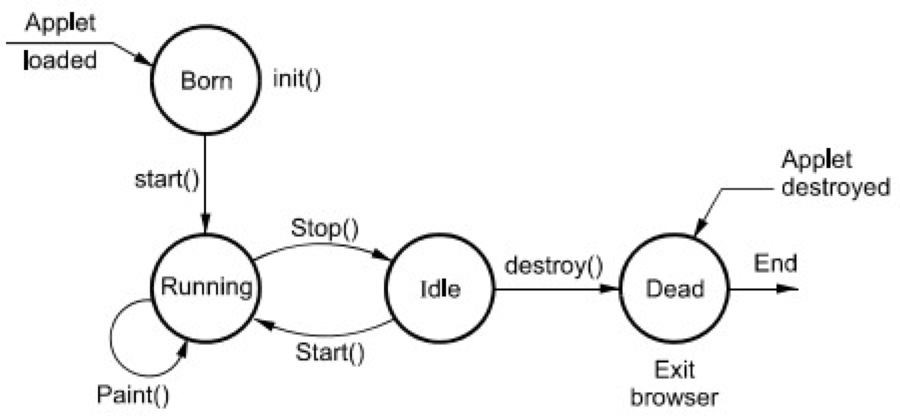


HelloApplet Started...

applet Stared

Life Cycle of an Applet
There are various methods which are typically used in applet for initialization and termination purpose. These methods are

1. Initialization
2. Running state
3. Idle state
4. Dead or destroyed state



When applet begins, the AWT calls following methods in sequence -

a) init() b) start() c) paint()

When applet is terminated following methods are invoked in sequence.

a) stop() b) destroy()

1. Initialization state

When applet gets loaded it enters in the initialization state. For this purpose the init() method is used. In this method you can initialize the required variables. This method is called only once initially at the execution of the program. The syntax can be

```
public void init()
{
//initialization of variables
}
```

In this method various tasks of initialization can be performed such as -

1. Creation of objects needed by applet
2. Setting up of initial values
3. Loading of image
4. Setting up of colors.

2. Running state

When the applet enters in the running state, it invokes the start() method of Applet class. This method is called only after the init method. After stopping the applet when we restart the applet at that time also this method is invoked. The syntax can be

```
public void start()
{
...
}
```

3. Display state

Applet enters in the display state when it wants to display some output. This may happen when applet enters in the running state. The paint() method is for displaying or drawing the contents on the screen. The syntax is

```
public void paint(Graphics g)
{
...
}
```

An instance of Graphics class has to be passed to this function as an argument. In this method various operations such as display of text, circle, line are invoked.

4. Idle state

This is an idle state in which applet becomes idle. The stop() method is invoked when we want to stop the applet .When an applet is running if we go to another page then this method is invoked. The syntax is

```
public void stop()
{
...
}
```

5. Dead state

When applet is said to be dead then it is removed from memory. The method destroy() is invoked when we want to terminate applet completely and want to remove it from the memory.

```
public void destroy()
{
...
}
```

Executing an Applet

There are two methods to run the applet

1. Using web browser 2. Using Appletviewer

Let us learn both the methods with necessary illustrations

1. Using web browser

Step 1 : Compile your Applet source program using javac compiler, i.e.

```
D:\test>javac FirstApplet.java
```

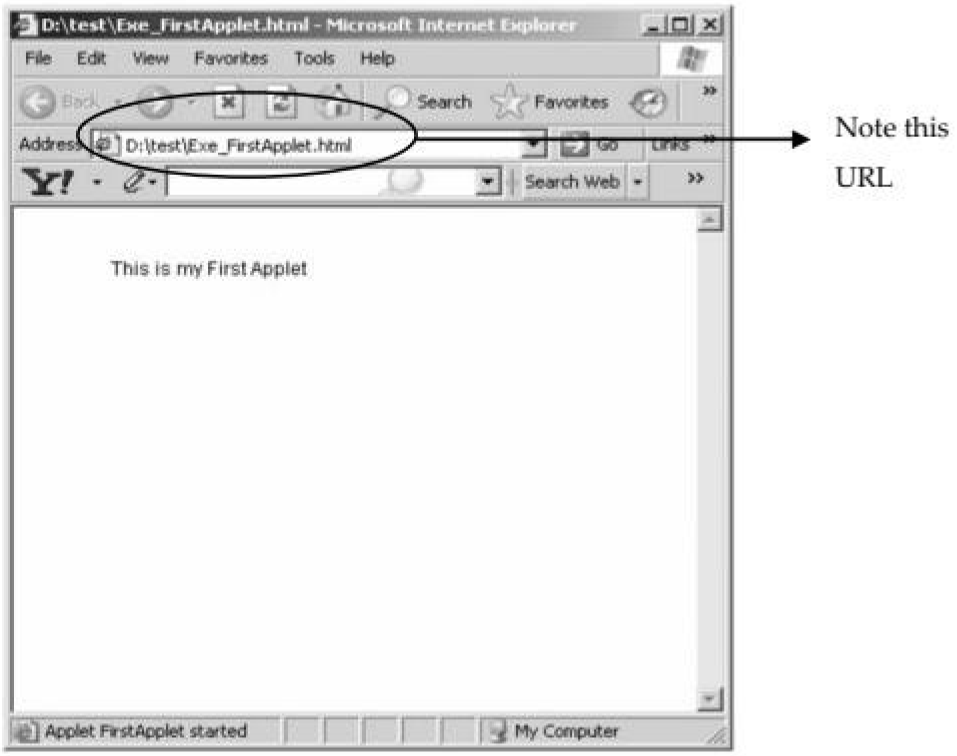
Step 2 : Write following code in Notepad/Wordpad and save it with filename and extension .html.

For example following code is saved as Exe_FirstApplet.html, The code is

<applet code="FirstApplet" width=300 height=100>

</applet>

Step 3 : Load html file with some web browser, This will cause to execute your html file. It will look this -


D:\test\Exe_FirstApplet.html - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Search Favorites
Address D:\test\Exe_FirstApplet.html
Go Links
Search Web
Note this URL
Applet FirstApplet started
My Computer

This is my First Applet

2. Using Appletviewer

Step 1 : To run the applet without making use of web browser or using command prompt we need to modify the code little bit. This modification is as shown below

/* This is my First Applet program */

import java.awt.*;

import java.applet.*;

/* <applet code="FirstApplet" width=300 height=100>

</applet> */

public class FirstApplet extends Applet

{

public void paint(Graphics g)

```
{
g.drawString("This is my First Applet",50,30);
}
}
```

In above code we have added one more comment at the beginning of the program

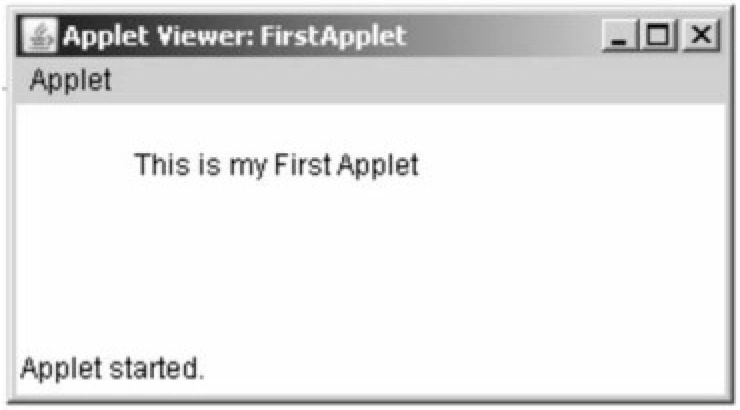
```
<applet code="FirstApplet" width=300 height=100>
</applet>
```

This will help to understand Java that the source program is an applet with the name FirstApplet. By this edition you can run your applet program merely by Appletviewer command.

Step 2 :

```
D:\test>javac FirstApplet.java
D:\test>Appletviewer FirstApplet.java
```

And we will get



```
This is my First Applet
Applet started.
```

Using Color in Applet

When you want to specify the color in order to make your applet colourful, there are some methods supported by AWT system. The syntax of specifying color is Color(int R,int G,int B);
Color(int RGBVal); //here RGBval denotes the value of color

Similarly to set the background color of applet window we use

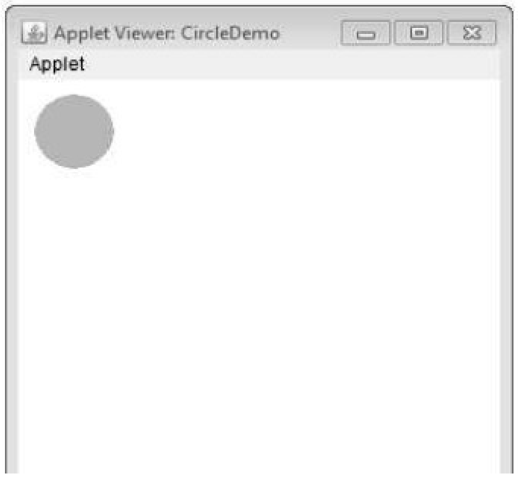
void setBackground(Color colorname)

where colorname denotes the name of the background color. In the same manner we can also specify the foreground color i.e. color of the text by using method

void setForeground(Color colorname)

Write an applet program for displaying the circle in green color.

```
import java.awt.*;
import java.applet.*;
/*
<applet code="CircleDemo" width=300 height=300>
</applet>
*/
public class CircleDemo extends Applet
{
public void paint(Graphics g)
{
g.setColor(Color.green);
g.fillOval(10,10,50,50);
}
}
```



Using Fonts in Applet

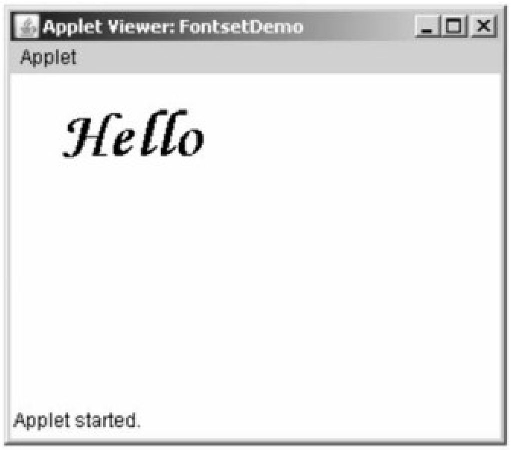
We can obtain the list of all the fonts that are present in our computer with the help of getAvailableFontFamilyNames(). This method is member of GraphicsEnvironment. First of all we need a reference to GraphicsEnvironment. For obtaining reference to GraphicsEnvironment we can call the getLocalGraphicsEnvironment( ) method. Then using this reference getAvailableFontFamilyNames() is invoked.

We can set the desired font using setFont() function. This function requires a reference parameter which can be created by the method Font(). In Font() method the first parameter is name of the font, second parameter denotes style of the font which can BOLD, ITALIC or PLAIN and third parameter denotes the size of font. Here is a simple program which is demonstrate the same

Java Program[FontsetDemo.java]

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

```
<applet code="FontsetDemo" width=310 height=200>
</applet>
*/
public class FontsetDemo extends Applet
{
String msg=" ";
public void init()
{
Font f;
f=new Font("Monotype Corsiva",Font.BOLD,40);
msg="Hello";
setFont(f);
}
public void paint(Graphics g)
{
g.drawString(msg,30,50);
}
}
```



Hello

Applet started.

Requesting the repaint() method

One of the important architectural constraints that have been imposed on an applet is that it must quickly return control to the AWT run-time system. It cannot create a loop inside paint( ). This would prevent control from passing back to the AWT. Whenever your applet needs to update the information displayed in its window, it simply calls ***repaint( )***. The ***repaint( )*** method is defined by the AWT that causes AWT run-time system to execute a call to your applet's ***update()*** method, which in turn calls ***paint()***. The AWT will then execute a call to paint( ) that will display the stored information. The repaint( ) method has four forms. The simplest version of repaint( ) is:

1. void repaint ( )

This causes the entire window to be repainted. Other versions that will cause repaint are:

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

If your system is slow or busy, update( ) might not be called immediately. If multiple calls have been made to AWT within a short period of time, then update( ) is not called very frequently. This can be a problem in many situations in which a consistent update time is necessary. One solution to this problem is to use the following forms of **repaint( ):**

3. void repaint (long **maxDelay)**

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

Where "maxDelay" is the number milliseconds should be elapsed before updat() method is called.

Using the Status Window

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

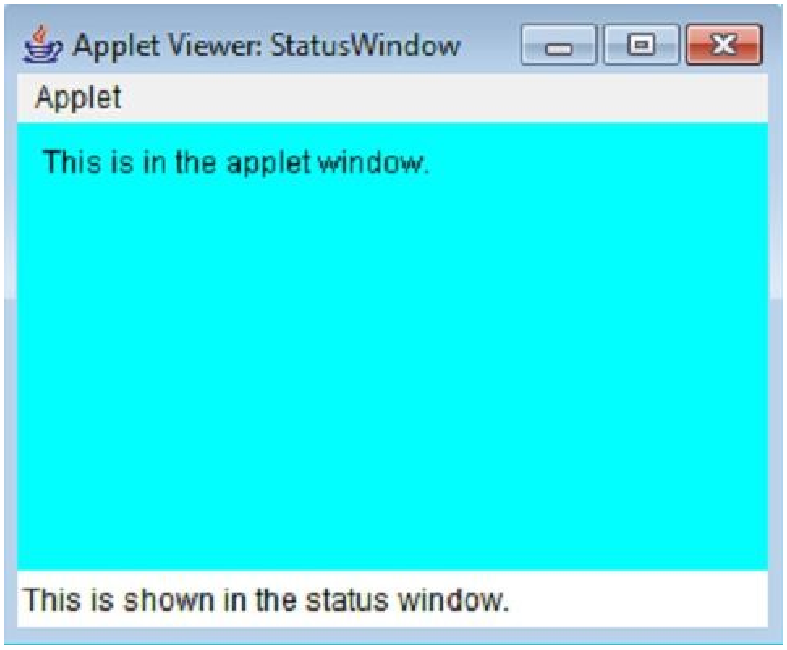
Example program

// Using the Status Window.

```
import java.awt.*;
import java.applet.*; /* StatusWindow.java */

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

Running applet:

1. javac StatusWindow.java
2. appletviewer StatusWindow.java



```
This is in the applet window.
This is shown in the status window.
```

Passing parameters to Applet

We can supply user defined parameters to an applet using the **<param>** Tag of HTML. Each <param> Tag has the attributes such as "name" , "value" to which actual values are assigned.

Using this tag we an change the text to be displayed through applet. We write the <param> Tag as follow:

<param   name="name1" value = "Hello Applet">   </param>

Passing parameters to an Applet is something 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> Tag in the HTML file
2. Provide the code in the applet to take these parameters.

ParaPassing.java

```
import java.applet.*;
import java.awt.*;
public class ParaPassing extends Applet
{
String str;
 public void init()
{
str=getParameter("name1");
if(str==null)
str="Java";
str="Hello "+str;
}
public void paint(Graphics g)
{
g.drawString(str,50,50);

 } }
```

para.html

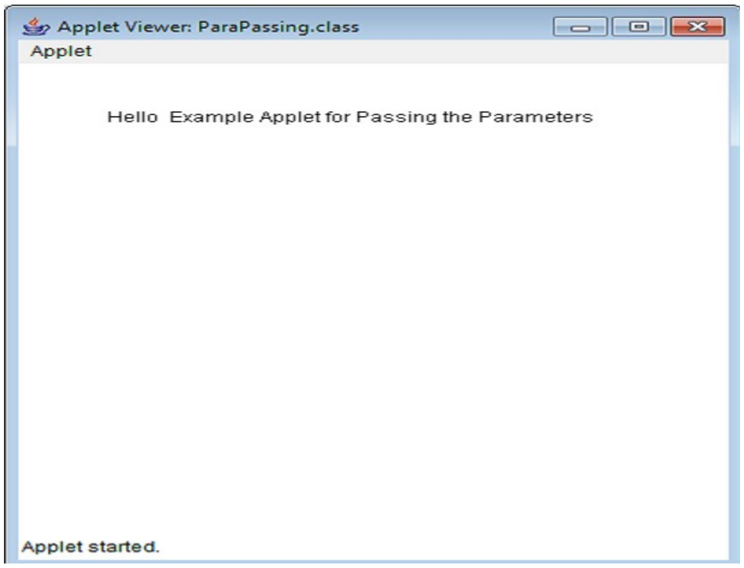
```
<html>
  <head>      <title> Passing Parameters</title> </head>
<body bgcolor=pink >
<applet code="ParaPassing.class" width=400 height=400 >
 <param name="name1" value="Example Applet for Passing the Parameters" >
</param>  </applet>
</body> </html>
```

Running the Program:

1. Compile the "ParaPassing.java" using the "javac" command, which generates the "ParaPassing.class " file

2. Use "ParaPassing.class" file to code attribute of the <applet> Tag and save it as "para.html"

3. Give "para.html" as input to the "**appletviewer**" to see the output or open the file using the applet enabled web browser.



```
Hello  Example Applet for Passing the Parameters
```

Introduction to Swing

- Swing is another approach of graphical programming in Java.
- Swing creates highly interactive GUI applications.
- It is the most flexible and robust approach.

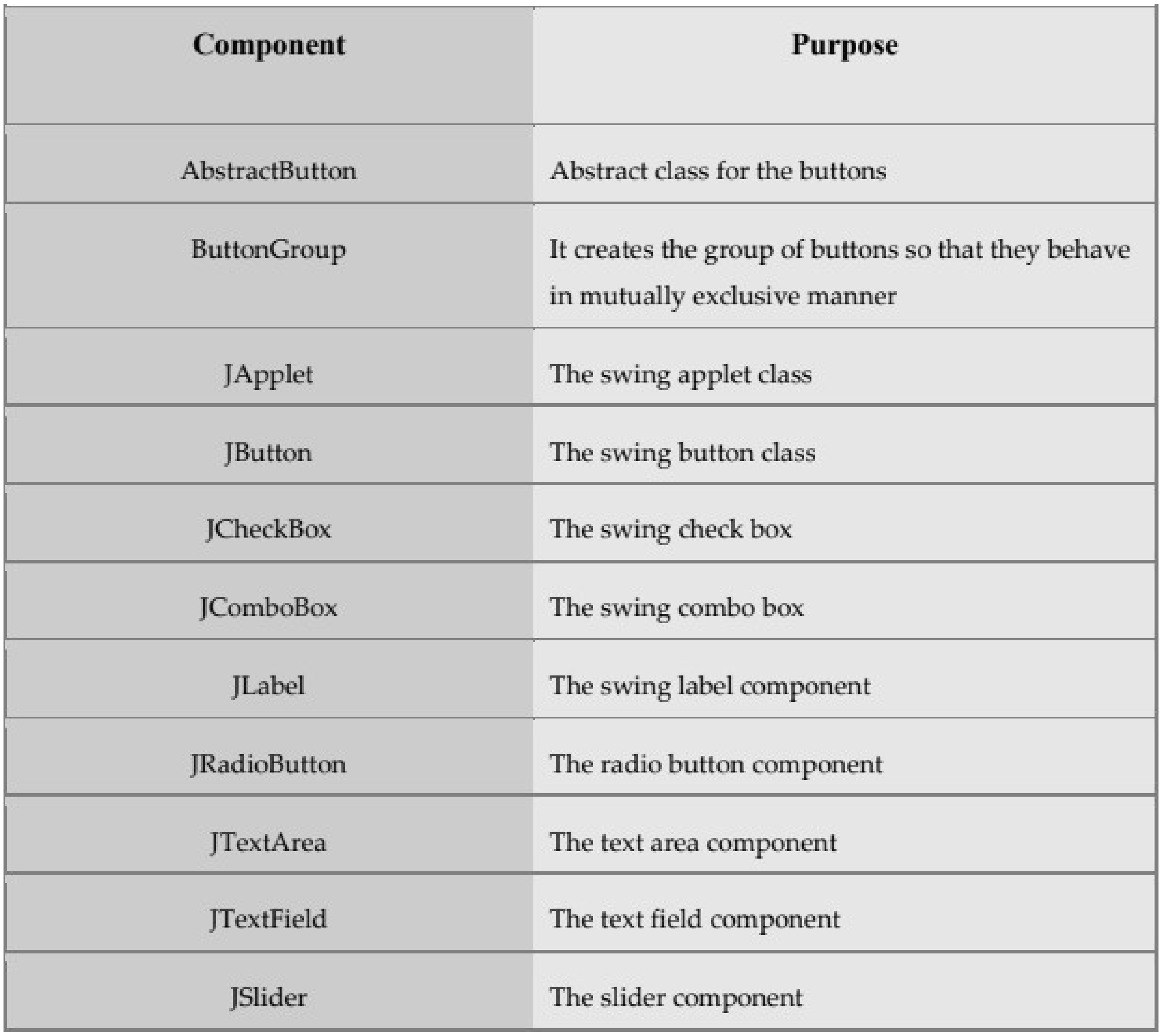
Difference between AWT and Swing

AWT	Swing
The Abstract Window ToolKit is a heavy weight component because every graphical unit will invoke the native methods.	The Swing is a light weight component because it's the responsibility of JVM to invoke the native methods.
The look and feel of AWT depends upon platform.	As Swing is based on Model View Controller pattern, the look and feel of swing components in independent of hardware and the operating

	system.
AWT occupies more memory space.	Swing occupies less memory space.
AWT is less powerful than Swing.	Swing is extension to AWT and many drawbacks of AWT are removed in Swing.

Swing Components

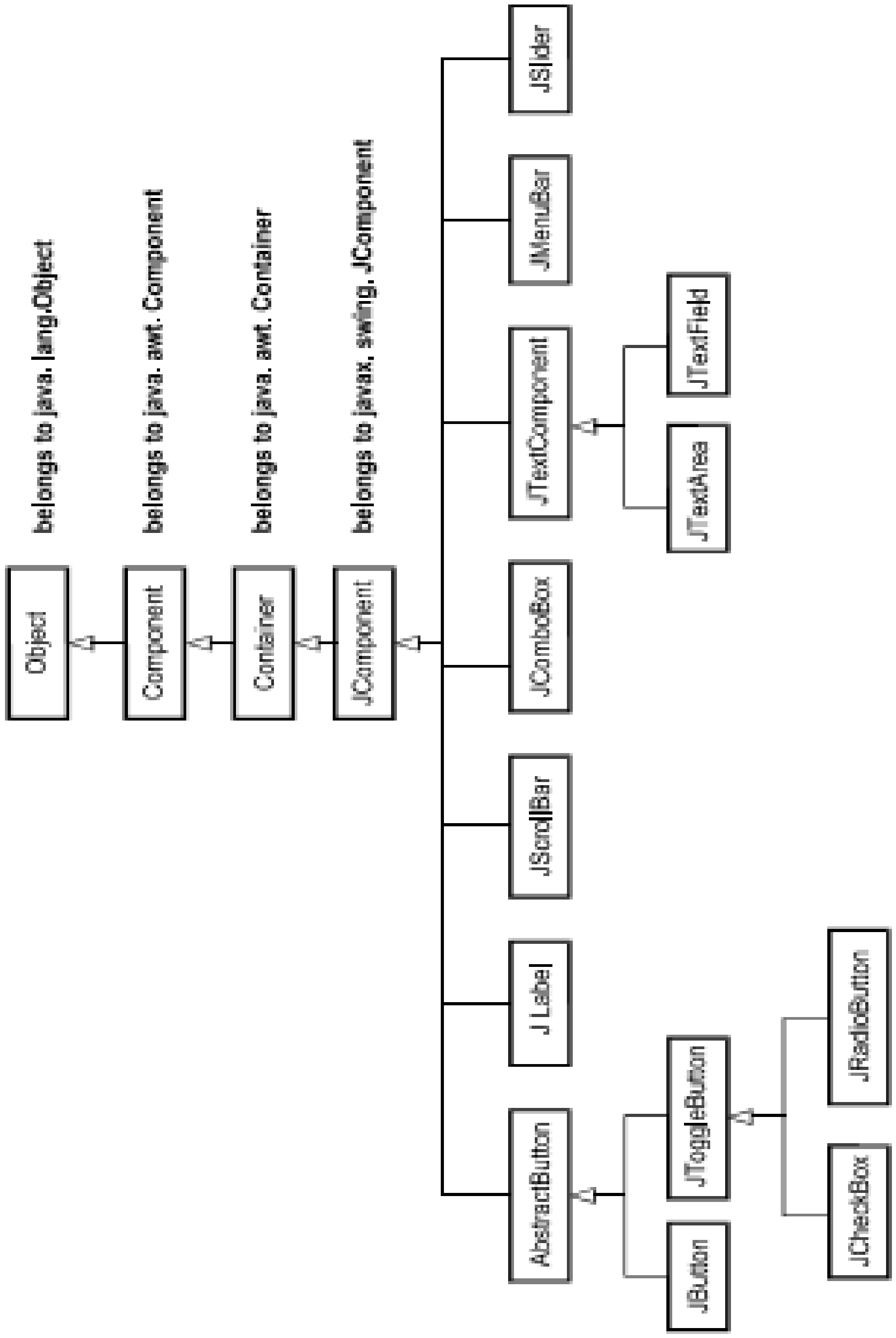
The most commonly used component classes are -



Component	Purpose
AbstractButton	Abstract class for the buttons
ButtonGroup	It creates the group of buttons so that they behave in mutually exclusive manner
JApplet	The swing applet class
JButton	The swing button class
JCheckBox	The swing check box
JComboBox	The swing combo box
JLabel	The swing label component
JRadioButton	The radio button component
JTextArea	The text area component
JTextField	The text field component
JSlider	The slider component

Swing Class Component Hierarchy

In order to display any JComponent on the GUI, it is necessary to add this component to the container first. If you do not add these components to container then it will not be displayed on the GUI. The swing class component hierarchy is



Layout Management

Definition :

- A Layout manager is an interface which automatically arranges the controls on the screen.
- Thus using layout manager the symmetric and systematic arrangement of the controls is possible. In this section we will discuss following layout managers.

FlowLayout

- FlowLayout manager is the simplest Layout manager.
- Using this Layout manager components are arranged from top left corner lying down from left to right and top to bottom.
- Between each component there is some space left.
- The syntax of FlowLayout manager is as given below -

FlowLayout(int alignment)

Where alignment denotes the alignment of the components on the applet windows. The alignment can be denoted as :

FlowLayout.LEFT

FlowLayout.RIGHT

FlowLayout.CENTER

BorderLayout

- In BorderLayout there are **four components** at the four sides and one component occupying large area at the centre.
- The central area is called CENTER and the components forming four sides are called LEFT,RIGHT,TOP and BOTTOM.

GridLayout

- GridLayout is a Layout manager used to arrange the components in a grid. The syntax of GridLayout manager is GridLayout(int n,int m) Where *n* represents total number of rows and *m* represents total number of columns.

CardLayout

- Sometimes we want to perform various sets of graphical controls at a time then CardLayout is used.
- Thus CardLayout manager allows us to have more than one layouts on the applet.
- The CardLayout is conceptually thought as a collection of cards lying on a **panel**.

We have to follow following steps -

Step 1 : We have to create two objects

1. Panel object 2. CardLayout object

Step 2 : Then we have to add the cards on the panel using **add()** method. For example - panel_obj.setLayout(layout_obj);

where *panel_obj* is an object of panel anf *layout_obj* is an object of CardLayout

Step 3 : Finally we have to add the object of panel to main applet. For example - add(panel_obj);

Event handlers

Event Classes

- Event classes are the classes responsible for handling events in the event handling mechanism.
- The **EventObject** class is at the top of the event class hierarchy. It belongs to the **java.util** package. And other event classes are present in **java.awt.event** package.
- The **getSource()** and **toString()** are the methods of the **EventObject** class.
- There is **getId()** method belonging to **java.awt.event** package returns the **nature of the event.**
- For example, if a keyboard event occurs, you can find out whether the event was key press or key release from the event object.

Various event classes that are defined in **java.awt.event** class are –

1. An **ActionEvent** object is generated when a component is activated. For example if a button is pressed or a menu item is selected then this event occurs.

2. An **AdjustmentEvent** object is generated when scrollbars are used.
3. A **TextEvent** object is generated when text of a component or a text field is changed.

4. A **ContainerEvent** object is generated when component are added or removed from container.
5. A **ComponentEvent** object is generated when a component is resized, moved, hidden or made visible.
6. An **ItemEvent** is generated when an item from a list is selected. For example a choice is made or if checkbox is selected.
7. A **FocusEvent** object is generated when component receives keyboard focus for input.
8. A **KeyEvent** object is generated when key on keyboard is pressed or released.
9. A **WindowEvent** object is generated when a window activated, maximized or minimized.
10. A **MouseEvent** object is generated when a mouse is clicked, moved, dragged, released.

Event Listeners

- The task of handling an event is carried out by **event listener**.
- When an event occurs, first of all an event object of the appropriate type is created. This object is then passed to a **Listener**.
- A listener must **implement the interface** that has the method for event handling.
- The **java.awt.event** package contains definitions of all event classes and **listener interface**.
- **An Interface** contains constant values and method declaration.
- The methods in an interface are only declared and not implemented, i.e. the methods do not have a body.
- The interfaces are used to define behavior on occurrence of event that can be implemented by any class anywhere in the class hierarchy.

Various event listener interfaces are -

1. ActionListener

This interface defines the method **actionPerformed()** which is invoked when an **ActionEvent** occurs. The **syntax** of this method is

void actionPerformed(ActionEvent act), where act is an object of ActionEvent class.

2. AdjustmentListener

This interface defines the method **adjustmentValueChanged()** which is invoked when an **AdjustmentEvent** occurs.

The syntax of this method is

void adjustmentValueChanged(ActionEvent act)

3. TextListener

It has a method **textChanged()** when a change in text area or text field occurs then this method is invoked.

void textChanged(TextEvent tx)

4. ContainerListener

When a component is added to container then this interface is required. There are two methods for this interface and those are -

void componentAdded(ContainerEvent ct)

void componentRemoved(ContainerEvent ct)

Where ct represents the object of class ContainerEvent

5. ComponentListener

When component is shown, hidden, moved or resized then the corresponding methods are defined by **ContainerListener i**nterface.

The **syntax** for the methods defined by this interface is

void componentShown(ComponentEvent co)

void componentHidden(ComponentEvent co)

void componentMoved(ComponentEvent co)

void componentResized(ComponentEvent co)

6. ItemListener

The **itemStateChanged()** is the only method defined by the **ItemListener** interface. The **syntax** is void itemStateChanged(ItemEvent It)

7. FocusListener

By this interface the methods related to keyboard focus are used. These methods are -

void focusGained(FocusEvent fo)

void focusLost(FocusEvent fo)

8. WindowFocusListener

By this interface the methods related to windows focus are used. These methods are - void windowGainedFocus(WindowEvent fo)

void windowLostFocus(WindowEvent fo) These methods are called when window gains or loses focus.

9. KeyListener

This interface is defining the events such as **keyPressed(),keyReleased() and keyTyped()** are used. These methods are useful for key press, key release and when you type some characters.

void keyPressed(keyEvent k) void keyReleased(keyEvent k) void keyTyped(keyEvent k)

10. MouseListener

This interface defines five important methods for various activities such as mouse click, press, released, entered or exited. These are

void mouseClicked(MouseEvent m)

void mousePressed(MouseEvent m)

void mouseReleased(MouseEvent m)

void mouseEntered(MouseEvent m)

void mouseExited(MouseEvent m)

11. MouseMotionListener

For handling mouse drag and mouse move events the required methods are defined by MouseMotionListener interface. These methods are

void mouseDragged(MouseEvent m)

void mouseMoved(MouseEvent m)

12. WindowsListener

There are seven methods in which are related to windows activation and deactivation.

void windowOpened(WindowEvent w)
void windowClosed(WindowEvent w)
void windowClosing(WindowEvent w)
void windowActivated(WindowEvent w)
void windowDeactivated(WindowEvent w)
void windowIconified(WindowEvent w)
void windowDeiconified(WindowEvent w)

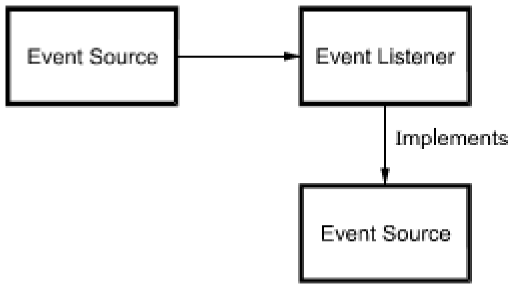
Relationship between Event Sources and Event Listener

An event source is an object that can register listener objects. And the listener object is an instance of a class that can implement a special interface called **listener interface**

Following are the steps that denote how event handling in AWT works -

Step 1 : The **event source**(Such as button or checkbox) sends the **event objects** to all the **registered listeners** when an event occurs.

Step 2 : The listener objects will then use the information in the event object and then determine reaction.

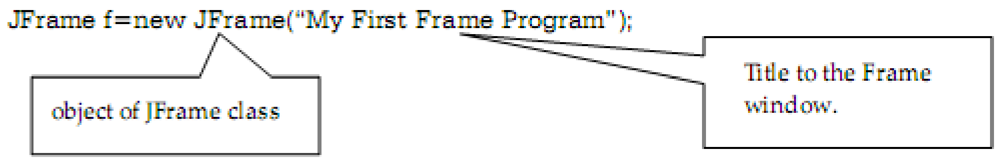


Mouse Events

While handling the mouse events we use two interfaces MouseListener and MouseMotionListener. These interfaces has certain methods such as mouseClicked(), mousePressed(), mouseReleased(), mouseEntered(), mouseExited() and mouseDragged() , mouseMoved() respectively.

Creating Frames

A Frame is a top-level window with a title and a border. The Frames in java works like the main window where the components like JButton,JLabel, JComboBox and so on can be placed. In Java swing the top-level windows are represented by the **JFrame** class. For creating the frame following statement can be used –


JFrame f=new JFrame("My First Frame Program");
object of JFrame class
Title to the Frame window.

- The frames are not visible initially, hence we have to make them visible by f.setVisible(true);
- The close button of the frame by default performs the hide operation for the JFrame. Hence we have to change this behavior to window close operation by setting the setDefaultCloseOperation() to **EXIT_ON_CLOSE** value.
- Any suitable size of the frame can be set by **setSize(int *height*,int *width*)** function.
- In the following program, we have created a frame on which two buttons and one textfield components are placed. The image icons are associated with the buttons. If the

button having “apple” image is pressed then the string “Apple” will be displayed in the text field and if the button having “orange” image is pressed then the string “Orange” will be displayed in the text field.

Label and Image Icon

- The icons are encapsulated by **ImageIcon** class. The constructors are - ImageIcon(string *fna me*)
- ImageIcon(URL *url*) The *fname* denotes the name of the file which contains the icon. The *url* denotes the resource of
- image.
- The **JLabel** is a component for placing the label component. The **JLabel** is a subclass of **JComponent** class. The syntax for the JLabel is - JLabel(Icon ic); Label(String s);

JLabel(String s, Icon ic, int alignment)

- The alignment can be LEFT,RIGHT,CENTER,LEADING and TRADING. The icons and text methods associated with the label are -
- Icon getIcon() void setIcon(Icon ic)<- is represents the icon file String getText(); void setText(String s);<-s represents the string

Text Field

- The **JTextField** is extended from the **JComponent** class. The **JTextField** allows us to add a single line text.
- The syntax of using **JTextField** is - JTextField();
- JTextField(int col_val); JTextField(String s,int col_val); JTextField(String s);

Text Area The JTextArea is a GUI control which allows us to write multi-line text. We can set the desired number of rows and number of columns so that a long big message can be written using this control.

- The syntax for JTextArea is

  JTextArea(String *str*,int *ro ws* ,int *co ls*);
- In the following Java program, there are two GUI controls on the window - the textarea and the push button. The idea is that: just type something within the text area and then click the push button.

Buttons

- The swing push button is denoted by using **JButton** class.
- The swing button class provides the facilities that can not be provided by the applet button class. For instance you can associate some image file with the button.
- The swing button classes are subclasses of **AbstractButton** class.
- The **AbstractButton** class generates action events when they are pressed. These events can be associated with the Push buttons.
- We can associate an icon and/or string with the **JButton** class.
  The syntax of JButton is
  JButton(Icon ic);
  JButton(String s);
  JButton(String s,Icon ic);

Checkboxes

- The Check Box is also implementation of **AbstractButton** class. But the immediate superclass of **JCheckBox** is **JToggleButton**.
- The JCheckBox supports two states true or false.
- We can associate an icon, string or the state with the checkboxes. The syntax for the Check Box will be -

JCheckBox(Icon ic);

JCheckBox(Icon ic, boolean state);

JCheckBox(String s);

JCheckBox(String s,boolean state);

JCheckBox(String s,Icon ic,boolean state);

Radio Buttons

- The JRadioButton is a subclass of **JToggleButton.** This control is similar to the checkboxes.
- The syntax for the Radio Box will be - JRadioButton(Icon ic);

JRadioButton (Icon ic, boolean state); JRadioButton (String s); JRadioButton (String s,boolean state); JRadioButton (String s,Icon ic,boolean state);

Lists

JList is a component that displays list of text items. Various components of JList component are

JList()	Creates a JList with an empty, read-only, model.
JList(ary[] listData)	Creates a JList that displays the elements in the specified array.
JList(ListModel<ary> dataModel)	Creates a JList that displays elements from the specified, non-null, model.

Various methods of this component are –

Method	**Description**
int getSelectedIndex()	It returns the index of selected item of the list.
void setListData(Object[] list)	It is used to create a read-only ListModel from an array of objects

Choices

- **JList** and **JCombobox** are very similar components but the **JList** allows to select multiple selections whereas the **JCombobox** allows only one selection at a time.
- A combo box is a combination of text field and the drop down list. The **JComboBox** is a subclass of **JComponent** class.

Scrollbars

- The scroll pane is a rectangular areas in which some component can be placed.
- The component can be viewed with the help of **horizontal** and **vertical** scroll bars.
- Using the **JScrollPane** class the component can be added in the program.

- The **JScrollPane** class extends the **JComponent** class.
- There are **three constructors** that can be used for this component –
  JScrollPane(Component *co mpo nen t*)
  JScrollPane(int *vs crollba r* , int *hs crollba r*)
  JScrollPane(Component *co mpo nen t*, int *vs c rollbar* , int *hs crollbar*)

The *component* represents the reference to the component. The *vscrollbar* and *hscrollbar* are the integer values for the vertical and horizontal scroll bars. These values can be defined by the constants such as

Constant	Meaning
HORIZONTAL_SCROLLBAR_ALWAYS	It always displays the horizontal scroll bar.
HORIZONTAL_SCROLLBAR_NEEDED	It displays the horizontal scrollbar if required.
VERTICAL_SCROLLBAR_ALWAYS	It always displays the vertical scroll bar.
VERTICAL_SCROLLBAR_NEEDED	It displays the vertical scrollbar if required.

Menus

- The menu bar is the most standard GUI which is present in almost all the applications.
- We can create a menu bar which contains several menus.
- Each menu can have several menu items.
- The separator can also be used to separate out these menus.
- Various methods and APIs that can be used for creating the Menus in Swing are -

JMenuBar

This class creates a menu bar. A menu bar normally contains several menus.

JMenu(String *str)*

This class creates several menus. The menus contain the menu items. The string *str* denotes the names of the menus.

JMenuItem(String *str*)

The menu items are the parts of menus. The string *str* denotes the names of the menu items.

setMnemonic(char *ch*)

The mnemonic key can be set using this method. The character which is passed to it as an argument becomes the mnemonic key. Hence using alt + ch you can select that particular menu.

JSeparator

This is the constructor of the class JSeparator which adds separating line between the menu items.

setJMenuBar

This method is used to set menu bar to a specific frame. In the following Java program, we have created a frame on which a menu bar is placed. The menu bar contains three menus - File, Edit and Help. Each of these menus have their own set of menu items. The separators and mnemonic keys are also used in this program.

Dialog boxes

> The dialog box is useful to display some useful messages to the user. Using Swing we can create three types of dialog boxes -

1. Simple message dialog box 2. Confirm message dialog box 3. Input dialog box

These dialog boxes are created using **JOptionPane** class. This class is present in the **javax.swing.*** package. Hence we must import this file at the beginning.

1. Simple message dialog box

- This type of message box simply displays the informative message to the user.
- It has only OK button.
- It is expected that after reading out the message the user must click the OK button to return.
- The **JOptionPane** class provides the method **showMessageDialog()** in order to display the simple message box.

2.Confirm message dialog box

- This type of dialog box will display some message to the user and will get the confirmation from him.
- User can give his confirmation either by clicking OK, CANCEL, YES or NO button.
- The confirm message dialog box will display the message either along with the OK and CANCEL button or with the YES, NO or CANCEL button.

JDBC Overview

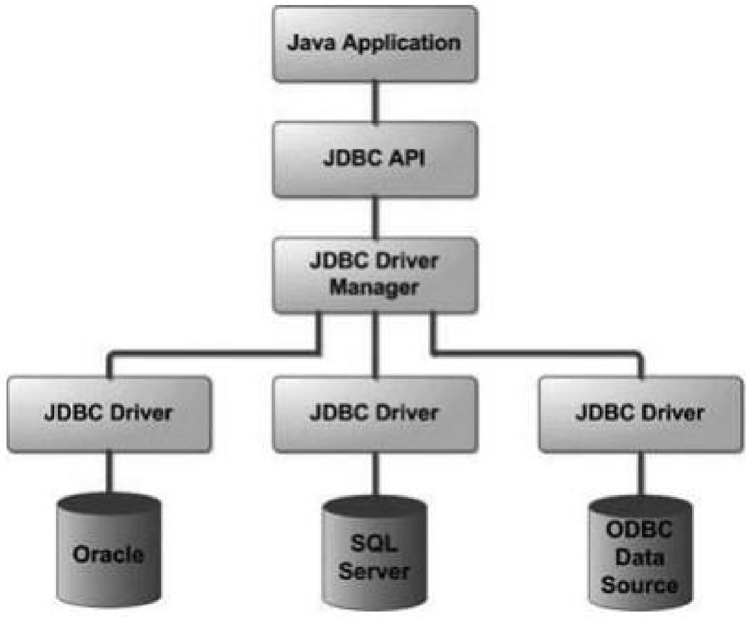
- ✓ JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database.
- ✓ JDBC works with Java on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.
- ✓ JDBC stands for Java Database Connectivity, which is a standard Java API for databaseindependent connectivity between the Java programming language and a wide range of databases.
- ✓ The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.
  - o Making a connection to a database.
  - o Creating SQL or MySQL statements.
  - o Executing SQL or MySQL queries in the database.
  - o Viewing & Modifying the resulting records.

Applications of JDBC

- ✓ Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as
  - o Java Applications
  - o Java Applets
  - o Java Servlets
  - o Java ServerPages (JSPs)
  - o Enterprise JavaBeans (EJBs).

JDBC Architecture

- ✓ The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers –
  - o **JDBC API: This provides the application-to-JDBC Manager connection.**
  - o **JDBC Driver API: This supports the JDBC Manager-to-Driver Connection.**
- ✓ The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.
- ✓ The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.
- ✓ Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application



JDBC Components

The JDBC API provides the following interfaces and classes -

DriverManager: This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.

Driver: This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.

Connection: This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.

Statement: You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.

ResultSet: These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.

SQLException: This class handles any errors that occur in a database application.

Creating JDBC Application

There are following six steps involved in building a JDBC application:

- ✓ **Import the packages:** Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.
- ✓ **Register the JDBC driver:** Requires that you initialize a driver so you can open a communication channel with the database.
- ✓ **Open a connection:** Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database.
- ✓ **Execute a query:** Requires using an object of type Statement for building and submitting an SQL statement to the database.
- ✓ **Extract data from result set:** Requires that you use the appropriate ResultSet.getXXX() method to retrieve the data from the result set.
- ✓ **Clean up the environment:** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

JDBC Connection Class

The programming involved to establish a JDBC connection is fairly simple. Here are these simple four steps:

- ✓ **Import JDBC Packages:** Add **import** statements to your Java program to import required classes in your Java code.
- ✓ **Register JDBC Driver:** This step causes the JVM to load the desired driver implementation into memory so it can fulfill your JDBC requests.
- ✓ **Database URL Formulation:** This is to create a properly formatted address that points to the database to which you wish to connect.
- ✓ **Create Connection Object:** Finally, code a call to the *DriverManager* object's *getConnection( )* method to establish actual database connection.

Import JDBC Packages

- ✓ The **Import** statements tell the Java compiler where to find the classes you reference in your code and are placed at the very beginning of your source code.
- ✓ To use the standard JDBC package, which allows you to select, insert, update, and delete data in SQL tables, add the following *imports* to your source code

```
import java.sql.* ;  // for standard JDBC programs
import java.math.* ; // for BigDecimal and BigInteger support
```

Register JDBC Driver

- ✓ You must register the driver in your program before you use it. Registering the driver is the process by which the Oracle driver's class file is loaded into the memory, so it can be utilized as an implementation of the JDBC interfaces.
- ✓ You need to do this registration only once in your program. You can register a driver in one of two ways.

1. Approach I - Class.forName()

- ✓ The most common approach to register a driver is to use Java's **Class.forName()** method, to dynamically load the driver's class file into memory, which automatically registers it.
- ✓ This method is preferable because it allows you to make the driver registration configurable and portable.

The following example uses Class.forName( ) to register the Oracle driver -

```
try {
  Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException ex) {
System.out.println("Error: unable to load driver class!");
System.exit(1); }
```

2. Approach II - DriverManager.registerDriver()

- ✓ The second approach you can use to register a driver, is to use the static **DriverManager.registerDriver()** method.
- ✓ You should use the registerDriver() method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

The following example uses registerDriver() to register the Oracle driver

```
try {
  Driver myDriver = new oracle.jdbc.driver.OracleDriver();
  DriverManager.registerDriver( myDriver );
}
catch(ClassNotFoundException ex) {
System.out.println("Error: unable to load driver class!");
System.exit(1); }
```

Database URL Formulation

- ✓ After loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method.

The following are list the three overloaded DriverManager.getConnection() methods:

- getConnection(String url)
- getConnection(String url, Properties prop)
- getConnection(String url, String user, String password)
- ✓ Here each form requires a database **URL**. A database URL is an address that points to your database.
- ✓ Formulating a database URL is where most of the problems associated with establishing a connection occurs.

Following table lists down the popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	**jdbc:mysql:**//hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:@**hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	**jdbc:db2:**hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	**jdbc:sybase:Tds:**hostname: port Number/databaseName

- ✓ All the highlighted part in URL format is static and you need to change only the remaining part as per your database setup.