COLORS IN JAVA

To support different colors Java package comes with the Color class. The Color class states colors in the default sRGB color space or colors in arbitrary color spaces identified by a ColorSpace.

Color class static color variables available are:

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

Color class constructor

Color(float r, float g, float b) – create color with specified red, green, and blue values in the range (0.0 - 1.0)

Color(int r, int g, int b)- create color with the specified red, green, and blue values in the range (0 - 255).

Some of the commonly used methods supported by the Color class are as follows.

Method	Description
int getRed()	Returns the red component in the range 0-255 in the default sRGB space.
int getGreen()	Returns the green component in the range 0-255 in the default sRGB space.
int getBlue()	Returns the blue component in the range 0-255 in the default sRGB space.
Color getHSBColor(float h, float s, float b)	Creates a Color object based on the specified values for the HSB color model.

The current graphics color can be changed using setColor() method defined in Graphics class.

```
void setColor(Color newColor)          // newColor indicates new drawing color
```

The current color detail can be obtained using getColor() method. Its syntax is.

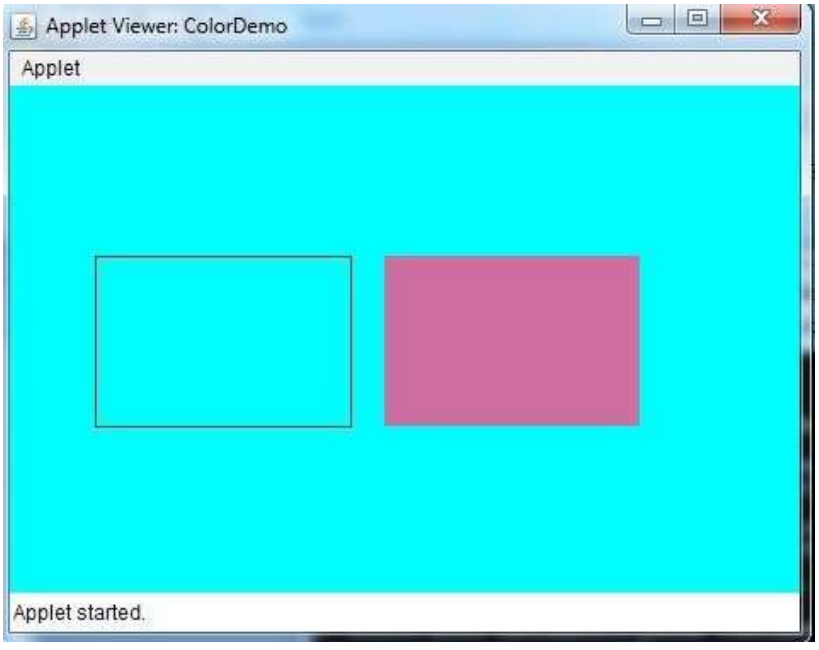
```
Color getColor()
```

Example:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="ColorDemo" width=350 height=300>
</applet>
*/
public class ColorDemo extends Applet {
```

```
    public void init() {
  setBackground(Color.CYAN);
  }
   public void paint(Graphics g) {
     g.setColor(Color.red);       // predefined color
     g.drawRect(50, 100, 150, 100); // rectangle outline is red color
     Color clr = new Color(200, 100, 150);
     g.setColor(clr);
     g.fillRect(220,100, 150, 100); // rectangle filled with clr color
  }
  }
```

Sample Output:



COMPONENTS

Java AWT Component classes exist in java.awt package. **The Component class is a super class of all components such as buttons, checkboxes, scrollbars, etc.**

Component class constructor:

Component() // constructs a new component

Properties of Java AWT Components:

- A Component object represents a graphical interactive area displayable on the screen that can be used by the user.
- Any subclass of a Component class is known as a component. For example, button is a component.
- Only components can be added to a container, like frame.

Some of the commonly used methods of Component class are as follows.

Method	Description
setBackground(Color)	Sets the background color of this component.
setBounds(int, int, int, int)	Moves and resizes this component.
setEnabled(boolean)	Enables or disables this component, depending on the value of the parameter b.
setFont(Font)	Sets the font of this component.
setForeground(Color)	Sets the foreground color of this component.
setLocation(int, int)	Moves this component to a new location.
setSize(int, int)	Resizes this component so that it has width width and height.
setVisible(boolean)	Shows or hides this component depending on the value of parameter b.
update(Graphics)	Updates this component.
repaint()	Repaints this component.
repaint(int, int, int, int)	Repaints the specified rectangle of this component.
add(Component c)	Inserts a component on this component.
remove(Component c)	Removes the specified component from this component.

Working with 2D shapes

Java supports 2-dimensional shapes, text and images using methods available in Graphics2D class. The Graphics2D class extends the Graphics class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout.

Graphics2D class Constructor

Graphics2D() //Constructs a new Graphics2D object.

This class inherits the methods from java.lang.Object. Some of the commonly used methods of Graphics2D class are as follows.

Method	Description
void draw(Shape s)	Strokes the outline of a Shape using the settings of the current Graphics2D context
void draw3DRect(int x, int y, int width, int height, boolean raised)	Draws a 3-D highlighted outline of the specified rectangle.
void drawImage(BufferedImage img, BufferedImageOp op, int x, int y)	Renders a BufferedImage that is filtered with a BufferedImageOp.
boolean drawImage(Image img, Affine Transform xform, ImageObserver obs)	Renders an image, applying a transform from image space into user space before drawing.
void drawString(String str, float x, float y)	Renders the text specified by the specified String, using the current text attribute state in the Graphics2D context
void fill(Shape s)	Fills the interior of a Shape using the settings of the Graphics2D context.
void rotate(double theta)	Concatenates the current Graphics2D Transform with a rotation transform.
void scale(double sx, double sy)	oncatenates the current Graphics2D Transform with a scaling transformation Subsequent rendering is resized according to the specified scaling factors relative to the previous scaling.
void setBackground(Color color)	Sets the background color for the Graphics2D context.
void setPaint(Paint paint)	Sets the Paint attribute for the Graphics2D context.
void setStroke(Stroke s)	Sets the Stroke for the Graphics2D context.
void shear(double shx, double shy)	Concatenates the current Graphics2D Transform with a shearing transform.
void transform(AffineTransform Tx)	Composes an AffineTransform object with the Transform in this Graphics2D according to the rulelast-specified-first-applied.
void translate(int x, int y)	Translates the origin of the Graphics2D con- text to the point (x, y) in the current coordi- nate system.

Example:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="ShapesDemo" width=350 height=300>
</applet>
*/
public class ShapesDemo extends Applet {
public void init() {}
```

```
public void paint(Graphics g) {
Graphics2D g2d = (Graphics2D)g;
g2d.setColor(Color.blue);
g2d.drawRect(75,75,300,200);
Font exFont = new
Font("TimesRoman",Font.PLAIN,40);
g2d.setFont(exFont);
g2d.setColor(Color.black);
g2d.drawString("Graphics2D
Example",120.0f,100.0f); g2d.setColor(Color.green);
g2d.drawLine(100,100,300,200);
g2d.drawOval(150,150,100,200);
g2d.fillOval(150,150,100,200);
}
}
```

Event Handling

Any change in the state of any object is called event. For Example: Pressing a button, en- tering a character in Textbox, Clicking or dragging a mouse, etc. The three main components in event handling are:

- *Events:* An event is a change in state of an object. For example, mouseClicked, mousePressed.
- *Events Source:* Event source is an object that generates an event. Example: a button, frame, textfield.
- *Listeners:* A listener is an object that listens to the event. A listener gets notified when an event occurs. When listener receives an event, it process it and then return. Listeners are group of interfaces and are defined in java.awt.event package. Each component has its own listener. For example MouseListener handles all MouseEvent.

Some of the event classes and Listener interfaces are listed below.

Event Classes	Generated when	Listener Interfaces
ActionEvent	button is pressed, menu-item is selected, list-item is double clicked	Action Listener
MouseEvent	mouse is dragged, moved, clicked, pressed or released and also when it enters or exit a component	Mouse Listener and Mouse Motion Listener
MouseWheelEvent	mouse wheel is moved	Mouse Wheel Listener
KeyEvent	input is received from keyboard	Key Listener
ItemEvent	check-box or list item is clicked	Item Listener
TextEvent	value of textarea or textfield is changed	Text Listener
AdjustmentEvent	scroll bar is manipulated	Adjustment Listener
WindowEvent	window is activated, deactivated, deico- nified, iconified, opened or closed	Window Listener
ComponentEvent	component is hidden, moved, resized or set visible	Component Listener
ContainerEvent	component is added or removed from container	Container Listener
FocusEvent	component gains or losses keyboard focus	Focus Listener

Java program for handling keyboard events.

Test.java

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

```
import java.awt.*;
//Implementing KeyListener interface to handle keyboard
events public class Test extends Applet implements
KeyListener
{
 String msg="";
 public void
 init()
 {
  addKeyListener(this);              //use keyListener to monitor key events
 }
 public void keyPressed(KeyEvent k)      // invoked when any key is pressed down
 {
  showStatus("KeyPressed");
 }
 public void keyReleased(KeyEvent k)    // invoked when key is released
 {
  showStatus("KeyRealesed");
 }
//keyTyped event is called first followed by key pressed or key released event
 public void keyTyped(KeyEvent k)        //invoked when a textual key is pressed
 {
  msg = msg+k.getKeyChar(); repaint();
 }
 public void paint(Graphics g)
 {
  g.drawString(msg, 20, 40);
 }
}
```

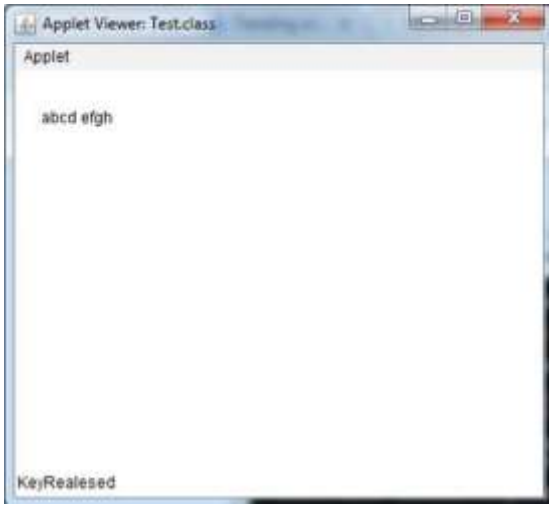
Test1.html

```
<html>
<body>
<applet code="Test.class" width="400" height="300">
```

</applet>

</body>

</html>



Adapter Classes

An adapter class provides the default implementation of all methods in an event listener interface. Adapter classes are very useful when you want to process only few of the events that are handled by a particular event listener interface. For example MouseAdapter provides empty implementation of MouseListener interface. It is useful because very often you do not really use all methods declared by interface, so implementing the interface directly is very lengthy.

- Adapter class is a simple java class that implements an interface with only EMPTY implementation.
- Instead of implementing interface if we extends Adapter class ,we provide implementation only for require method

The adapter classes are found in **java.awt.event**, **java.awt.dnd** and **javax.swing. event** packages. The Adapter classes with their corresponding listener interfaces are as fol- lows.

Adapter Class	Listener Interface
Window Adapter	Window Listener
Key Adapter	Key Listener
Mouse Adapter	Mouse Listener
Mouse Motion Adapter	Mouse Motion Listener
Focus Adapter	Focus Listener
Component Adapter	Component Listener
Container Adapter	Container Listener
HierarchyBoundsAdapter	HierarchyBoundsListener

Example:

```
import java.awt.*;
import
java.awt.event.*;
public class
   AdapterExample{ Frame f;
   AdapterExample(){
      f=new Frame("Window Adapter");
      f.addWindowListener(new WindowAdapter(){
         public void windowClosing(WindowEvent e) {
            f.dispose();
          }
       });

      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
   }
public static void main(String[] args)
   { new AdapterExample();
}
}
```

Sample Output:

Actions

The Java Action interface and AbstractAction class are terrific ways of

encapsulating be- haviors (logic), especially when an action can be triggered from more than one place in your Java/Swing application.

javax.swing

Interface Action

An Action can be used to separate functionality and state from a component. For example, if you have two or more components that perform the same function, consider using an Action object to implement the function.

An Action object is an action listener that provides not only action-event handling, but also centralized handling of the state of action-event-firing components such as tool bar buttons, menu items, common buttons, and text fields. The state that an action can handle includes text, icon, mnemonic, enabled, and selected status.

The most common way an action event can be triggered from multiple places in a Java/ Swing application is through the Java menubar (JMenuBar) and toolbar (JToolBar)

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
 public class ButtonAction {
    private static void createAndShowGUI()  {
       JFrame frame1 = new JFrame("JAVA Program");
      frame1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JButton button = new JButton(" << Java Action >>");
      //Add action listener to button
      button.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent e)
        {
           System.out.println("You clicked the button");
        }
      });
       frame1.getContentPane().add(button);
      frame1.pack();
      frame1.setVisible(true);
    }
     public static void main(String[] args) {
      javax.swing.SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            createAndShowGUI();
}
    } });
```

FONTS IN JAVA

The Font class states fonts, which are used to render text in a visible way.

Font class constructor

Font(Font font) //Creates a new Font from the specified font.

Font(String name, int style, int size) //Creates a new Font from the specified name, style and point size.

Font variables available in Font class are:

Font.BOLD	Font. SANS_SERIF
Font.ITALIC	Font. CENTER_BASELINE
Font. PLAIN	Font. DIALOG
Font. MONOSPACED	Font. SERIF
Font. TRUETYPE_FONT	Font. TYPE1_FONT
int size	int style
float pointSize	String name

Some of the commonly used methods supported by the Font class are as follows.

Method	Description
String getFamily()	Returns the family name of this Font.
int getStyle()	Returns the style of this Font.
boolean isBold()	Indicates whether or not this Font object's style is BOLD
boolean isItalic()	Indicates whether or not this Font object's style is ITALIC.
boolean isPlain()	Indicates whether or not this Font object's style is PLAIN.
static Font getFont(String nm)	Returns a Font object fom the system properties list.
static Font decode(String str)	Returns the Font that the str argument describes.
String toString()	Converts this Font object to a String representation.

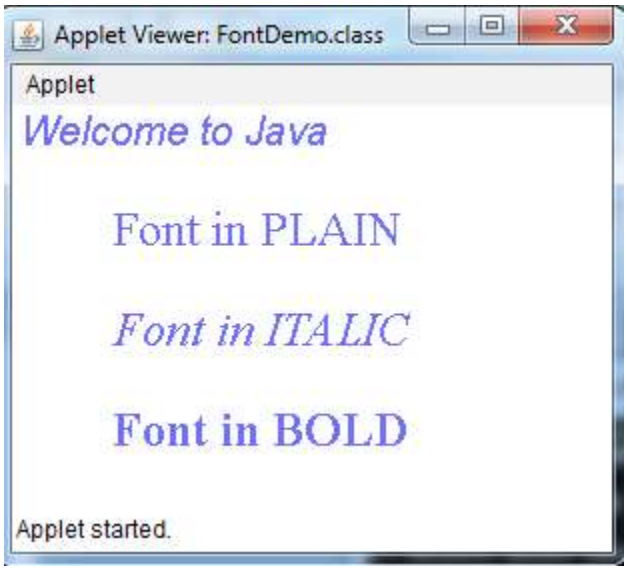
Example:

```
import
java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/* <APPLET CODE =”FontDemo.class” WIDTH=300 HEIGHT=200> </APPLET> */
public class FontDemo extends java.applet.Applet
  {
    Font f;
    String
    m;
     public void init()
       {
          f=new Font(“Arial”,Font.ITALIC,20);
```

```
        m="Welcome to Java";
        setFont(f);
    }
        public void paint(Graphics g)
          {
            Color c=new
            Color(100,100,255);
            g.setColor(c);
            g.drawString(m,4,20);
Font plainFont = new Font("Serif", Font.PLAIN, 24);

g.setFont(plainFont);
 g.drawString("Font in PLAIN", 50,
 70);
 Font italicFont = new Font("Serif", Font.ITALIC, 24);
 g.setFont(italicFont);
 g.drawString("Font in ITALIC", 50, 120);
 Font boldFont = new Font("Serif", Font.BOLD,
 24); g.setFont(boldFont);
 g.drawString("Font in BOLD", 50, 170);
 Font boldItalicFont = new Font("Serif", Font.BOLD+Font.ITALIC, 24);
 g.setFont(boldItalicFont);
 g.drawString("Font in BOLD ITALIC", 50, 220);
          }
 }
```

Sample Output



FRAMES

A Frame is a top-level window with a title and a border. Frames are capable of generating the following types of window events: WindowOpened, WindowClosing, WindowClosed, WindowIconified, WindowDeiconified, WindowActivated, WindowDeactivated.

Frame Constructor

Frame()

Constructs a new instance of `Frame` that is initially invisible.

Frame(String)

Constructs a new, initially invisible `Frame` object with the specified title.

Some of the commonly used methods of Frame class are as follows.

Methods	Description
String getTitle()	Gets the title of the frame.
void setBackground(Color bgColor)	Sets the background color of this window.
void setResizable (boolean resizable)	Sets whether this frame is resizable by the user.
void setShape (Shape shape)	Sets the shape of the window.
void setTitle (String title)	Sets the title for this frame to the specified string.
void setSize (Dimension d)	Resizes this component so that it has width d. width and height d.height.
void setVisible(boolean b)	Shows or hides this Window depending on the value of parameter b.
public void show()	Makes the Window visible
void setMenuBar (MenuBar) mb)	Sets the menu bar for this frame to the specified menu bar

Creating a Frame

We can generate a window by creating an instance of Frame. The created frame can be made visible by calling setVisible( ). When created, the window is given a default height and width. The size of the window can be changed explicitly by calling the setSize( ) method. A

label can be added to the current frame by creating an Label instance and calling the add() method.

Example:

```
import java.awt.*; public

class AwtFrame{

public static void main(String[] args){

Frame frm = new Frame("Java AWT Frame");

Label lbl = new Label("Welcome",Label.CENTER);

frm.add(lbl);

frm.setSize(400,400);

frm.setVisible(true);

 }

}
```


Sample Output:

Creating an Frame Window in an Applet

The steps to be followed to create a child frame within an applet are as follows.

1. Create a subclass of Frame

2. Override any of the standard window methods, such as init(),start(),stop(),and paint().
3. Implement the windowClosing() method of the windowListener interface,calling setVisible(false) when the window is closed
4. Once you have defined a Frame subclass, you can create an object of that class. But it will note be initially visible
5. When created, the window is given a default height and width
6. You can set the size of the window explicitly by calling the setSize() method

Example:

AppletFrame.java

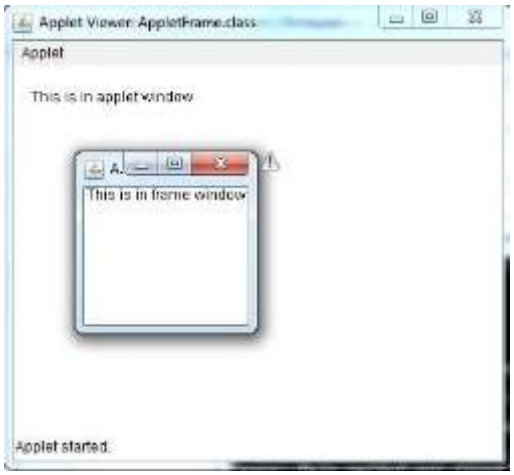
```
// Create a child frame window from within an applet.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
// Create a subclass of Frame.
class SampleFrame extends Frame {
SampleFrame(String title) {
super(title);
// create an object to handle window events
MyWindowAdapter adapter = new MyWindowAdapter(this);
// register it to receive those events
addWindowListener(adapter);
}
public void paint(Graphics g) { g.drawString("This
is in frame window", 10, 40);
}
}
class MyWindowAdapter extends WindowAdapter {
SampleFrame sampleFrame;
public MyWindowAdapter(SampleFrame sampleFrame) {
this.sampleFrame = sampleFrame;
}
public void windowClosing(WindowEvent we) {
sampleFrame.setVisible(false);
}
}
// Create frame window.
```

```
public class AppletFrame extends Applet { Frame
f;
//init(), start(), paint(), and stop() methods are called automatically in the specified se-
quence.
public void init() {
f = new SampleFrame("A FrameWindow");
f.setSize(150, 150);
f.setVisible(true);
}
public void start() {
f.setVisible(true); // make the window visible
}
public void stop() { f.setVisible(false); //
hide the window
}
public void paint(Graphics g) {
g.drawString("This is in applet window", 15, 30); // Display the given text in the win- dow
}
}
```


Test1.html

```
<html>
<body>
<applet code="AppletFrame.class" width="400" height="300">
</applet>
</body>
</html>
```

Sample Output:



IMAGES IN JAVA

Image control is superclass for all image classes representing graphical images.

Image class constructor

Image() // create an Image object

Some of the commonly used methods supported by the Image class are as follows.

Method	Description
Graphics getGraphics()	Creates a graphics context for drawing to an off-screen image.
int getHeight(ImageObserver observer)	Determines the height of the image.
Image getScaledInstance(int width, int height, int hints)	Creates a scaled version of this image.
ImageProducer getSource()	Gets the object that produces the pixels for the image.
int getWidth(ImageObserver observer)	Determines the width of the image.

The java.applet.Applet class provides following methods to access image.

1. getImage() method that returns the object of Image. Its syntax is as follows.

   public Image getImage(URL u, String image){}

2. getDocumentBase() method returns the URL of the document in which applet is embedded.

   public URL getDocumentBase(){}

3. URL getCodeBase() method returns the base URL.

   public URL getCodeBase()

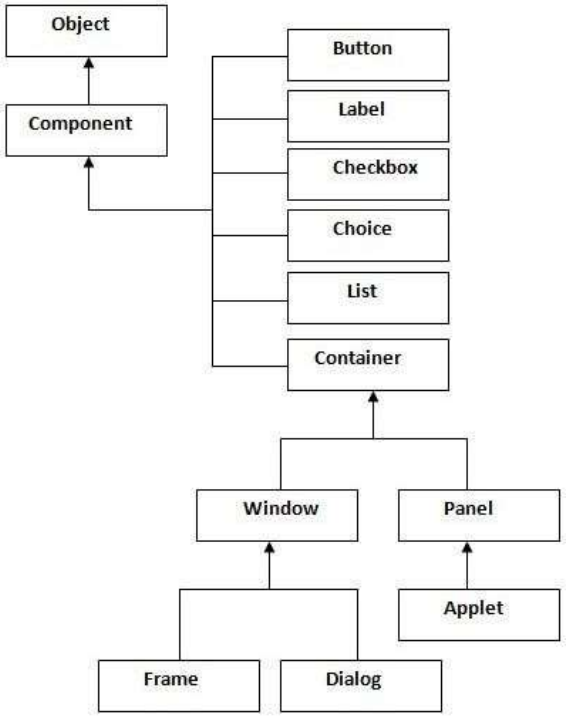
Example:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.net.URL;
/* <APPLET CODE =”ImageDemo.class” WIDTH=300 HEIGHT=200> </APPLET> */
public class ImageDemo extends java.applet.Applet
  {
    Image img; public
     void init()
       {
        }
          public void paint(Graphics g)
            {
```

```
        URL url1 = getCodeBase();
        img = getImage(url1,"java.jpg");
        g.drawImage(img, 60, 120, this);
}}
```

JAVA AWT HIERARCHY

The hierarchy of Java AWT classes are given below.



Container

The Container is a component in AWT that can contain another component like buttons, textfields, labels etc. The classes that extend Container class are known as container such as Frame, Dialog and Panel.

Window

The window is the container that has no borders and menu bars. You must use frame, dialog or another window for creating a window.

Panel

The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

The following table gives the methods of Component class:

Method	Description
public void add(Component c)	inserts a component on this component.

public void setSize(int width,int height)	sets the size (width and height) of the component.
public void setLayout(Layout Manager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

The following programs are the examples of Java AWT:

To create simple awt program, we need to create a frame. There are two ways to create a frame in AWT.

- By extending Frame class (inheritance)
- By creating the object of Frame class (association)

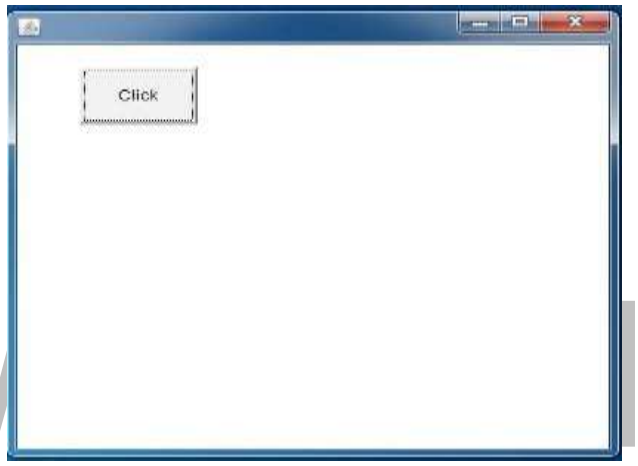
Example program using by extending Frame class (inheritance)

```
import java.awt.*;
class First extends Frame{
First(){
Button b=new Button("click me");
b.setBounds(30,100,80,30);// setting button position
add(b);//adding button into frame
setSize(300,300);//frame size 300 width and 300 height
setLayout(null);//no layout manager
setVisible(true);//now frame will be visible, by default not visible
}
public static void main(String args[]){
First f=new First();
}}
```

Example program using by creating the object of Frame class (association)

```
import java.awt.*;
class First2{
```

```
First2(){
Frame f=new Frame();
Button b=new Button("click me");
b.setBounds(30,50,80,30); f.add(b);
f.setSize(300,300);
f.setLayout(null);
f.setVisible(true);
}
public static void main(String args[]){
First2 f=new First2();
}}
```

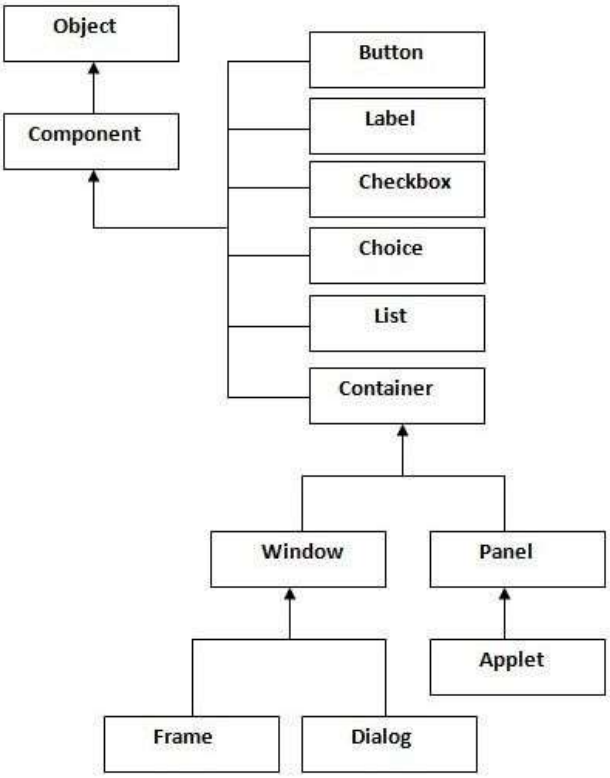


Java AWT

The **Abstract Window Toolkit (AWT) is Java's original platform-independent windowing, graphics, and user-interface widget toolkit.** The AWT classes are contained in the java.awt package.

- Contains all of the classes for creating user interfaces and for painting graphics and images.
- *an API to* ***develop GUI or window-based applications*** in java.

The hierarchy of Java AWT classes are shown below.



Component

A component is an object having a graphical representation that can be displayed on the screen and that can interact with the user.

Examples :

buttons, checkboxes, and scrollbars

The Component class is the abstract superclass of all user interface elements that are displayed on the screen. A Component object remembers current text font, foreground and background color.

Container

The Container class is the subclass of Component. The container object is a component that can contain other AWT components. It is responsible for laying out any components that it contains.

Window

The class Window is a top level window with no border and no menubar. The default lay- out for a window is `BorderLayout`. A window must have either a frame, dialog, or another window defined as its owner when it's constructed.

Panel

The class Panel is the simplest container class. It provides space in which an application can attach any other component, including other panels. The default layout manager for a panel is the FlowLayout layout manager

Frame

A Frame is a top-level window with a title and a border. It uses BorderLayout as default layout manager.

Dialog

A Dialog is a top-level window with a title and a border that is typically used to take some form of input from the user.

Canvas

A Canvas component represents a blank rectangular area of the screen onto which the application can draw or from which the application can trap input events from the user. An application must subclass the Canvas class in order to get useful functionality such as creat- ing a custom component. The paint method must be overridden in order to perform custom graphics on the canvas. It is not a part of hierarchy of Java AWT.

java.awt.Graphics class

The java.awt.Graphics class provides many methods for graphics programming. A graph- ics context is encapsulated by the Graphics class and is obtained in two ways:

- It is passed to an applet when one of its various methods, such as paint( ) or update( ) is called.
- It is returned by the getGraphics( ) method of Component.

Graphics Methods

The commonly used methods of Graphics class are as follows

Method	Description
abstract Graphics create()	Creates a new Graphics object that is a copy of this Graphics object
abstract void drawString(String str, int x, int y)	Draws the text given by the specified string
void drawRect(int x, int y, int width, int height)	draws a rectangle with the specified width and height
void draw3DRect(int x, int y, int width, int height, boolean raised)	Draws a 3-D highlighted outline of the specified rectangle.
abstract void drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)	Draws an outlined round-cornered rect- angle using this graphics context's current color

abstract void fillRect(int x, int y, int width, int height)	fill rectangle with the default color and specified width and height.
abstract void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)	Draws a closed polygon defined by arrays of x and y coordinates.
abstract void fillPolygon(int[] xPoints, int[] yPoints, int nPoints)	Fills a closed polygon defined by arrays of x and y coordinates.
abstract void drawOval(int x, int y, int width, int height)	draw oval with the specified width and height.
abstract void fillOval(int x, int y, int width, int height)	fill oval with the default color and specified width and height.
abstract void drawLine(int x1, int y1, int x2, int y2)	draw line between the points(x1, y1) and (x2, y2).
abstract boolean drawImage(Image img, int x, int y, ImageObserver observer)	draw the specified image.
abstract void drawArc(int x, int y, int width, int height, int startAngle, int arc Angle)	draw a circular or elliptical arc.
abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)	fill a circular or elliptical arc.
abstract void setColor(Color c)	set the graphics current color to the specified color.
abstract void setFont(Font font)	set the graphics current font to the specified font.

Example:

GraphicsDemo.java import

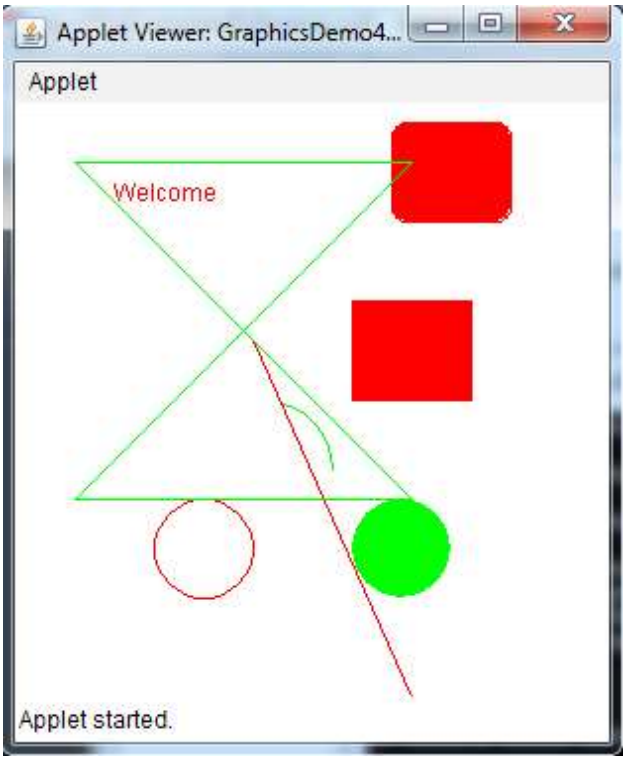
```
java.applet.Applet;  import
java.awt.*;
public class GraphicsDemo extends Applet{ public
void paint(Graphics g){ g.setColor(Color.red); //
set font color g.drawString("Welcome",50, 50); //
display text
g.drawLine(120,120,200,300);                    // draw a line
// draw and fill rectangle
g.drawRect(170,100,60,50);
g.fillRect(170,100,60,50);
// draw and fill rounded rectangle
g.drawRoundRect(190, 10, 60, 50, 15, 15);
g.fillRoundRect(190, 10, 60, 50, 15, 15);
// draw and fill oval
```

```
g.drawOval(70,200,50,50);
g.setColor(Color.green);
g.fillOval(170,200,50,50);
// draw and fill arc
g.drawArc(90,150,70,70,0,75);
g.fillArc(270,150,70,70,0,75);
// draw a polygon
int xpoints[] = {30, 200, 30, 200, 30};
int ypoints[] = {30, 30, 200, 200, 30};
int num = 5;
g.drawPolygon(xpoints, ypoints, num);
}
}
```

Test.html

```
<html>
<body>
<applet code="GraphicsDemo4.class" width="300" height="300">
</applet>
</body>
</html>
```


Sample Outpu



JAVA SWING

Swing was developed to provide a more sophisticated set of GUI components than the earlier Abstract Window Toolkit (AWT). Swing provides a look and feel that emulates the look and feel of several platforms, and also supports a pluggable look and feel that allows applications to have a look and feel unrelated to the underlying platform. It has more powerful and flexible components than AWT.

In addition to familiar components such as buttons, check boxes and labels, Swing provides several advanced components such as tabbed panel, scroll panes, trees, tables, and lists.

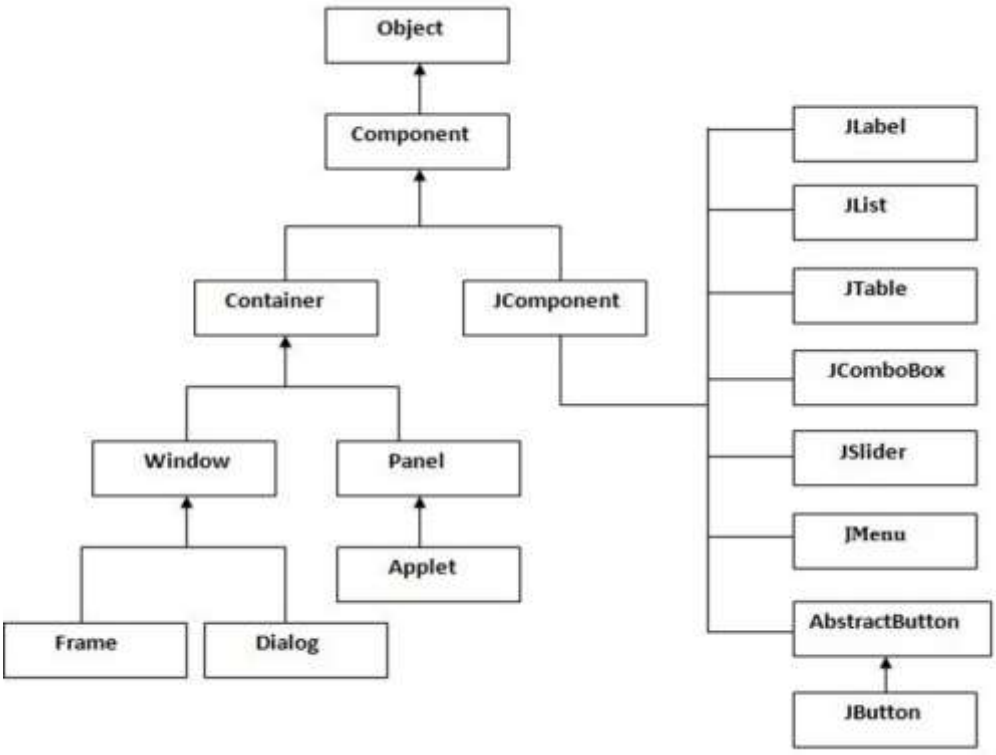
Unlike AWT components, Swing components are not implemented by platform-specific code. Instead, they are written entirely in Java and therefore are platform-independent. The term "lightweight" is used to describe such an element.

Java Swing is a part of Java Foundation Classes (JFC) that **is *used to create window-based applications*.** It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components. The

javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

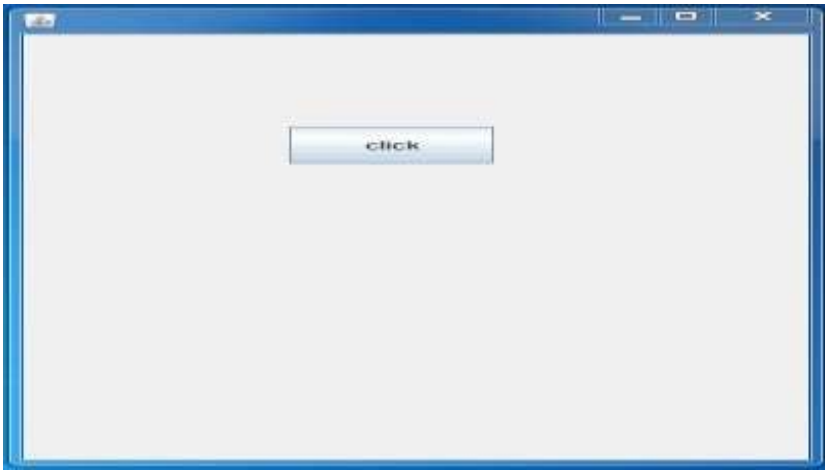
The hierarchy of java swing API is given below



The following program is an example for Java Swing.

```
import javax.swing.*;
public class FirstSwingExample { public
static void main(String[] args) {
JFrame f=new JFrame();//creating instance of JFrame
Button b=new JButton("click");//creating instance of JButton
b.setBounds(130,100,100, 40);//x axis, y axis, width, height
f.add(b);//adding button in JFrame
 f.setSize(400,500);//400 width and 500 height
f.setLayout(null);//using no layout managers
f.setVisible(true);//making the frame visible
}
  }
```

Output:



MouseEvent:

An event which indicates that a mouse action occurred in a component. A mouse action is considered to occur in a particular component if and only if the mouse cursor is over the unobscured part of the component's bounds when the action happens. For lightweight components, such as Swing's components, mouse events are only dispatched to the component if the mouse event type has been enabled on the component.

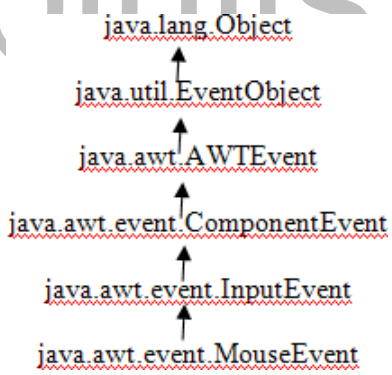
A mouse event type is enabled by adding the appropriate mouse-based EventListener to the component (Mouse Listener or Mouse Motion Listener), or by invoking Component.enableEvents (long) with the appropriate mask parameter

(AWTEvent.MOUSE_EVENT_MASK or AWTEvent.MOUSE_MOTION_EVENT_MASK).

If the mouse event type has not been enabled on the component, the corresponding mouse events are dispatched to the first ancestor that has enabled the mouse event type.Iif a MouseListener has been added to a component, or enableEvents(AWTEvent.MOUSE_EVENT_MASK) has been invoked, then all the events defined by MouseListener are dispatched to the component.

On the other hand, if MouseMotionListener has not been added and enableEvents has not been invoked with AWTEvent.MOUSE_MOTION_EVENT_MASK, then mouse motion events are not dispatched to the component. Instead the mouse motion events are dispatched to the first ancestor that has enabled mouse motion events.

The hierarchy of MouseEvent class is shown below.



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

A MouseEvent object is passed to every MouseListener or MouseAdapter object which is

registered to receive the "interesting" mouse events using the component's addMouseListener method. The MouseAdapter objects implement the MouseListener interface. Each such listener object gets a MouseEvent containing the mouse event.

A MouseEvent object is also passed to every MouseMotionListener or MouseMotionAdapter object which is registered to receive mouse motion events using the component's addMouseMotionListener method. (MouseMotionAdapter objects implement the MouseMotionListener interface.) Each such listener object gets a MouseEvent containing the mouse motion event.

When a mouse button is clicked, events are generated and sent to the registered MouseListeners. The state of modal keys can be retrieved using InputEvent.getModifiers() and InputEvent.getModifiersEx(). The button mask returned by InputEvent.getModifiers() reflects only the button that changed state, not the current state of all buttons.. To get the state of all buttons and modifier keys, use InputEvent.getModifiersEx(). The button which has changed state is returned by getButton().

For example, if the first mouse button is pressed, events are sent in the following order:

id modifiers button

MOUSE_PRESSED: BUTTON1_MASK BUTTON1

MOUSE_RELEASED: BUTTON1_MASK BUTTON1

MOUSE_CLICKED: BUTTON1_MASK BUTTON1

When multiple mouse buttons are pressed, each press, release, and click results in a separate event.

For example, if the user presses button 1 followed by button 2, and then releases them in the same order, the following sequence of events is generated:

id modifiers button

MOUSE_PRESSED: BUTTON1_MASK BUTTON1

MOUSE_PRESSED: BUTTON2_MASK BUTTON2

MOUSE_RELEASED: BUTTON1_MASK BUTTON1

MOUSE_CLICKED: BUTTON1_MASK BUTTON1

MOUSE_RELEASED: BUTTON2_MASK BUTTON2

MOUSE_CLICKED: BUTTON2_MASK BUTTON2

If **button 2** is released first, the MOUSE_RELEASED/MOUSE_CLICKED pair for BUTTON2_MASK arrives first, followed by the pair for BUTTON1_MASK.

MOUSE_DRAGGED events are delivered to the Component in which the mouse button was pressed until the mouse button is released (regardless of whether the mouse position is within the bounds of the Component). Due to platform-dependent Drag&Drop implementations, MOUSE_DRAGGED events may not be delivered during a native Drag&Drop operation.

In a multi-screen environment mouse drag events are delivered to the Component even if the mouse position is outside the bounds of the Graphics Configuration associated with that Component. However, the reported position for mouse drag events in this case may differ from the actual mouse position:

- Inamulti-screenenvironmentwithoutavirtualdevice:Thereportedcoordinatesformouse drag events are clipped to fit within the bounds of the GraphicsConfiguration associated with the Component.
- In a multi-screen environment with a virtual device: The reported coordinates for mouse drag events are clipped to fit within the bounds of the virtual device associated with the Component.

The following program is an example for MouseEvent.

```
import java.awt.*; import
java.awt.event.*;
        pubic class MouseListenerExample extends Frame implements MouseListener{
Label l;
MouseListenerExample(){
  addMouseListener(this);
  l=new Label();
  l.setBounds(20,50,100,20);
  add(l);
  setSize(300,300);
  setLayout(null);
  setVisible(true);
}
public void mouseClicked(MouseEvent e) {
  l.setText("Mouse Clicked");
}
public void mouseEntered(MouseEvent e) {
  l.setText("Mouse Entered");
}
public void mouseExited(MouseEvent e) {
  l.setText("Mouse Exited");
}
public void mousePressed(MouseEvent e) {
  l.setText("Mouse Pressed");
}
public void mouseReleased(MouseEvent e) {
  l.setText("Mouse Released");
}
public static void main(String[] args) {
```

```
    new MouseListenerExample();
  }
  }
```

Output: