Java Programming Tutorial

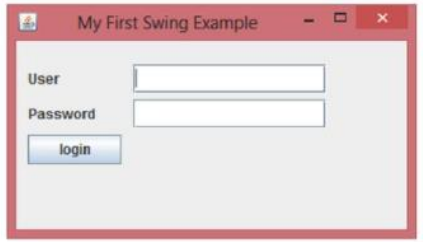
Programming Graphical User Interface (GUI)

https://www3.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI.html

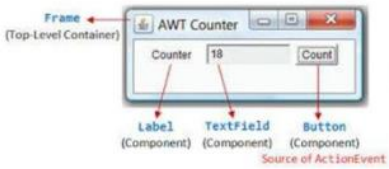
Introduction

JavaFX example

Swing example



AWT example



JavaFX vs. Swing & AWT

- Initially Java came with the AWT (Abstract Windows Toolkit) GUI (Graphical User Interface) library. AWT is fine for developing simple graphical user interfaces, but not for developing comprehensive GUI projects. AWT components are platform-dependent
- AWT was replaced by a more robust, versatile, and flexible library, Swing, in 1998. Swing components are platform-independent
- Java 8 introduced in 2014 the JavaFX GUI framework for developing Rich Internet Applications (RIA) that provide a desktop-like experience on Web applications. JavaFX is an excellent example of how the object-oriented principles are applied. JavaFX components are platform-independent

https://slideplayer.com/slide/16543324/

AWT Packages

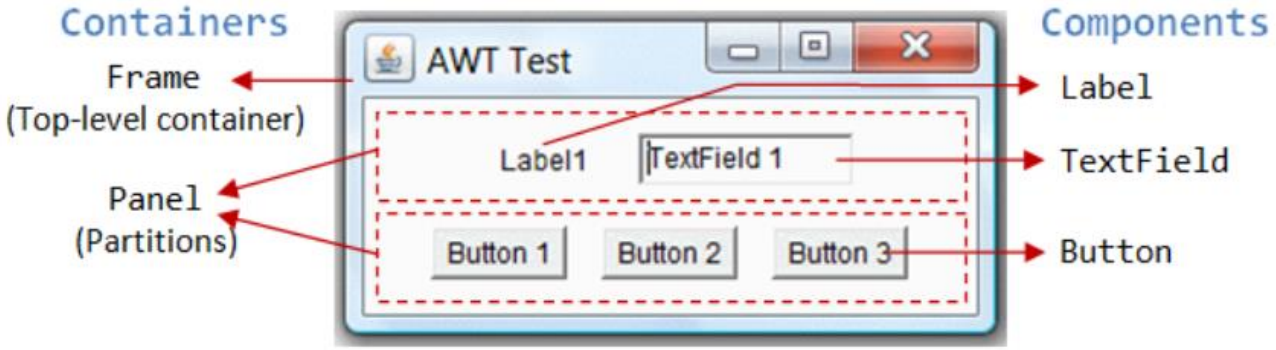
- AWT is huge! It consists of 12 packages of 370 classes (Swing is even bigger, with 18 packages of 737 classes as of JDK 8). Fortunately, only 2 packages - java.awt and java.awt.event - are commonly-used.
  - The java.awt package contains the core AWT graphics classes:
    - GUI Container classes, such as Frame and Panel.
    - Layout managers, such as FlowLayout, BorderLayout and GridLayout.
    - GUI Component classes, such as Button, TextField, and Label.
    - Custom graphics classes, such as Graphics, Color and Font.

AWT Packages

- The java.awt.event package supports event handling:
  - Event Listener Interfaces, such as ActionListener, MouseListener, MouseMotionListener, KeyListener and WindowListener,
  - Event classes, such as ActionEvent, MouseEvent, KeyEvent and WindowEvent,
  - Event Listener Adapter classes, such as MouseAdapter, KeyAdapter, and WindowAdapter.
- AWT provides a platform-independent and device-independent interface to develop graphic programs that runs on all platforms, including Windows, macOS, and Unixes.

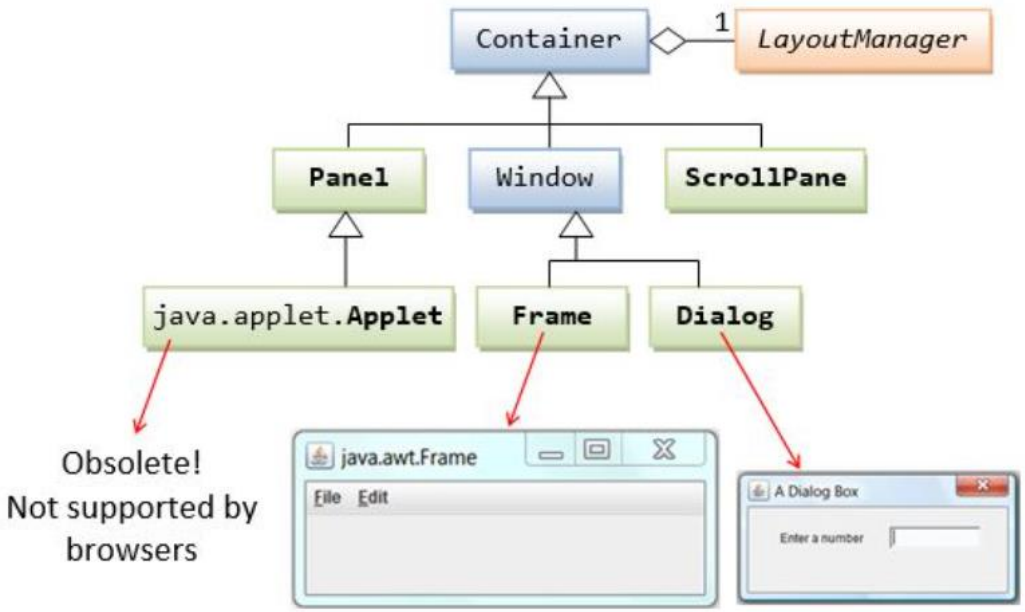
AWT Containers and Components

- There are two groups of GUI elements:
  - Component (Widget, Control): Components are elementary GUI entities, such as Button, Label, and TextField. They are also called widgets, controls in other graphics systems.
  - Container: Containers, such as Frame and Panel, are used to hold components in a specific layout (such as FlowLayout or GridLayout). A container can also hold sub-containers.
- In the above figure, there are three containers:
  a Frame and two Panels.
- In a GUI program, a component must be kept (or added) in a container. You need to identify a container to hold the components. Every container has a method called add(Component c).



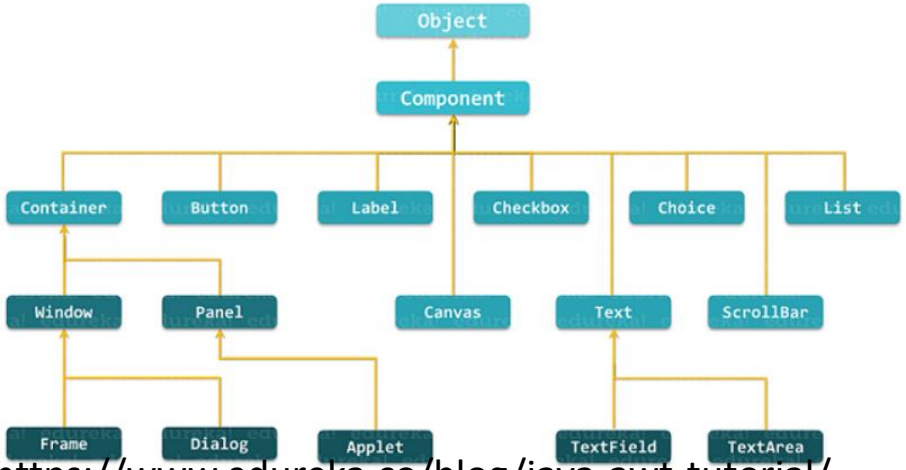
AWT Container Classes

- Top-Level Containers: Frame, Dialog and Applet
  - Each GUI program has a top-level container. The commonly-used top-level containers in AWT are Frame, Dialog and Applet
- Secondary Containers: Panel and ScrollPane
  - AWT provides these secondary containers:
  - Panel: a rectangular box used to layout a set of related GUI components in pattern such as grid or flow.
  - ScrollPane: provides automatic horizontal and/or vertical scrolling for a single child component.
  - others.
- A Container has a LayoutManager to layout the components in a certain pattern, e.g., flow, grid.

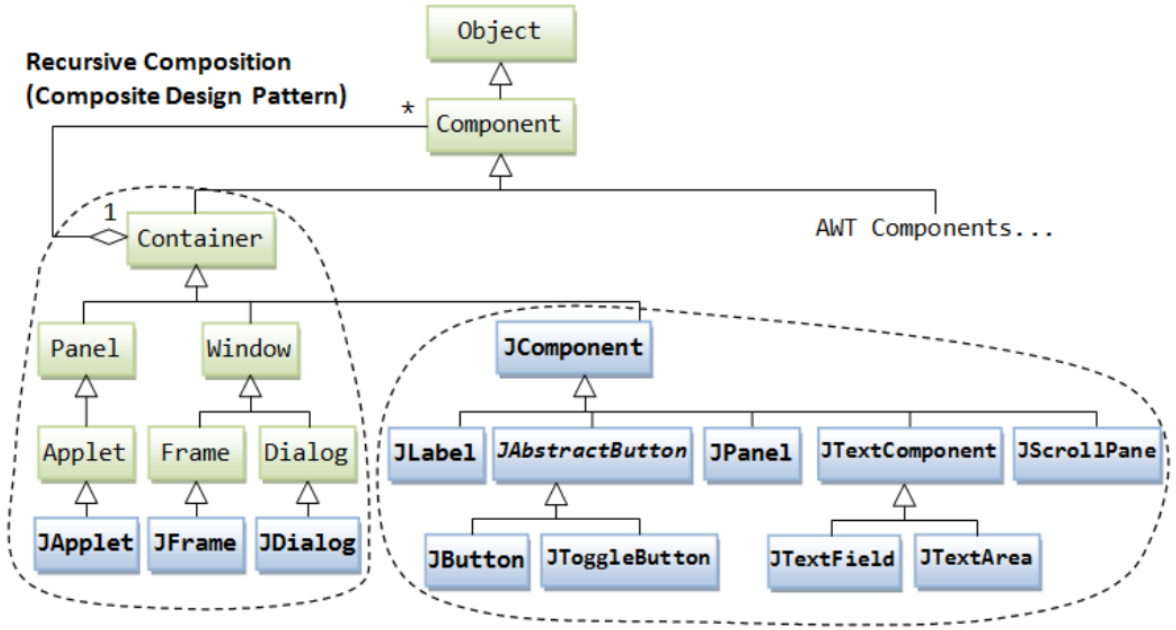


AWT Component Classes vs. Swing API

Hierarchy Of AWT



https://www.edureka.co/blog/java-awt-tutorial/

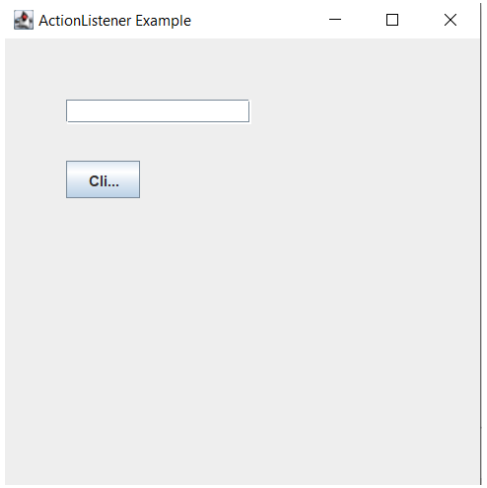


(Very Brief) Swing

- Swing application uses AWT event-handling classes (in package java.awt.event).
- Swing application uses AWT's layout manager
- Swing implements double-buffering and automatic repaint batching for smoother screen repaint.

AWT Event-Handling

- In event-driven programming, a piece of event-handling codes is executed (or called back by the graphics subsystem) when an event was fired in response to a user input (such as clicking a mouse button or hitting the ENTER key in a text field).
- the method actionPerformed() is known as a callback method. In other words, you never invoke actionPerformed() in your codes explicitly. The actionPerformed() is called back by the graphics subsystem under certain circumstances in response to certain user actions.
- The AWT's event-handling classes are kept in package java.awt.event.
- Three kinds of objects are involved in the event-handling: a source, listener(s) and an event object.




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

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;

//1st step : create an actionListener
public class Demo1 implements
        ActionListener{
    static private JFrame f;
    static private JTextField tf;
    static private JButton b;
```

```
public static void main(String[] args) {
  Demo1 demo = new Demo1();
  f= new JFrame("ActionListener Example");
  tf=new JTextField();
  tf.setBounds(50,50, 150,20);
  b=new JButton("Click Here");
  b.setBounds(50,100,60,30);
  //2nd step : register
  b.addActionListener(demo);
  f.add(b);f.add(tf);
  f.setSize(400,400);
  f.setLayout(null);
  f.setVisible(true);
  f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
//3rd step : define the handler
public void actionPerformed(ActionEvent e){
  tf.setText("will show number of clicks!");
}
}
```

Inner class

- A nested class (or commonly called inner class) is a class defined inside another class.
- A nested class has these properties:
  - A nested class is a proper class. That is, it could contain constructors, member variables and member methods. You can create an instance of a nested class via the new operator and constructor.
  - A nested class is a member of the outer class, just like any member variables and methods defined inside a class.
  - Most importantly, a nested class can access the private members (variables/methods) of the enclosing outer class, as it is at the same level as these private members. This is the property that makes inner class useful.
  - A nested class can have private, public, protected, or the default access, just like any member variables and methods defined inside a class. A private inner class is only accessible by the enclosing outer class, and is not accessible by any other classes. [An top-level outer class cannot be declared private, as no one can use a private outer class.]
  - A nested class can also be declared static, final or abstract, just like any ordinary class.
  - A nested class is NOT a subclass of the outer class. That is, the nested class does not inherit the variables and methods of the outer class. It is an ordinary self-contained class. [Nonetheless, you could declare it as a subclass of the outer class, via keyword "extends OuterClassName", in the nested class's definition.]

Properties of Anonymous Inner Class

1. The anonymous inner class is define inside a method, instead of a member of the outer class (class member). It is local to the method and cannot be marked with access modifier (such as public, private) or static, just like any local variable of a method.
2. An anonymous inner class must always extend a superclass or implement an interface. The keyword "extends" or "implements" is NOT required in its declaration. An anonymous inner class must implement all the abstract methods in the superclass or in the interface.
3. An anonymous inner class always uses the default (no-arg) constructor from its superclass to create an instance. If an anonymous inner class implements an interface, it uses the java.lang.Object().
4. An anonymous inner class is compiled into a class named OuterClassName$n.class, where n is a running number of inner classes within the outer class.
5. An instance of an anonymous inner class is constructed via this syntax:

```
new SuperClassName/InterfaceName() {  // extends superclass or implements interface
                                      // invoke the default no-arg constructor or Object[]
  // Implement abstract methods in superclass/interface
  // More methods if necessary
  ......
}
```

```java
//1st step : create an actionListener
public class Demo1 implements ActionListener{
  static private JFrame f;
  static private JTextField tf;
  static private JButton b;
  public static void main(String[] args) {
    Demo1 demo = new Demo1();
    f= new JFrame("ActionListener Example");
    tf=new JTextField();
    tf.setBounds(50,50, 150,20);
    b=new JButton("Click Here");
    b.setBounds(50,100,60,30);
    b.addActionListener(demo); //2nd step : register
    f.add(b);f.add(tf);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
  //3rd step : define the handler
  public void actionPerformed(ActionEvent e){
    tf.setText("want to show number of click here!");
  }
}
```

```java
public class Demo2 {
  private JFrame f;
  private JTextField tf;
  private JButton b;
  Demo2() {
    f= new JFrame("ActionListener by
AnonymousClass");
    tf=new JTextField();
    tf.setBounds(50,50, 150,20);
    b=new JButton("Click Here");
    b.setBounds(50,100,60,30);
    //2nd step : register
    Lis lis = new Lis();
    b.addActionListener( lis );
    f.add(b);f.add(tf);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
  class Lis implements ActionListener {
    public void actionPerformed(ActionEvent ev) {
      tf.setText("will show number of clicks!");
    }
  }
  public static void main(String[] args) { Demo2 demo = new Demo2();}
}
```

```java
public class Demo3 extends JFrame {
  private JTextField tf;
  private JButton b;
  Demo3() {
    setTitle("ActionListener by Anonymous");
    tf=new JTextField();
    tf.setBounds(50,50, 150,20);
    b=new JButton("Click Here");
    b.setBounds(50,100,60,30);
    //2nd step : register
    b.addActionListener( new ActionListener() {
      @Override //3rd step : define the handler
      public void actionPerformed(ActionEvent evt) {
          tf.setText("will show number of clicks!");
      }
      });
    add(b); add(tf);
    setSize(400,400);
    setLayout(null);
    setVisible(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
  public static void main(String[] args) { Demo3 demo = new Demo3(); }
}
```

```java
public class Demo4 extends JFrame {
  private JTextField tf;
  private JButton b;
  Demo3() {
    setTitle("ActionListener by Anonymous");
    tf=new JTextField();
    tf.setBounds(50,50, 150,20);
    b=new JButton("Click Here");
    b.setBounds(50,100,60,30);
    //2nd step : register
    b.addActionListener( ev
       -> tf.setText("will show number of clicks! ");
    add(b); add(tf);
    setSize(400,400);
    setLayout(null);
    setVisible(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
  public static void main(String[] args) { Demo3 demo = new Demo3(); }
}
```

Listeners

- WindowEvent and WindowListener Interface
- MouseEvent and MouseListener Interface
- MouseEvent and MouseMotionListener Interface
- KeyEvent and KeyListener Interface

```
public void mouseClicked(MouseEvent evt)
   // Called-back when the mouse-button has been clicked (pressed followed by released) on the source.
public void mousePressed(MouseEvent evt)
public void mouseReleased(MouseEvent evt)
   // Called-back when a mouse-button has been pressed/released on the source.
   // A mouse-click invokes mousePressed(), mouseReleased() and mouseClicked().
public void mouseEntered(MouseEvent evt)
public void mouseExited(MouseEvent evt)
   // Called-back when the mouse-pointer has entered/exited the source.
```

```
public void mouseDragged(MouseEvent e)
   // Called-back when a mouse-button is pressed on the source component and then dragged.
public void mouseMoved(MouseEvent e)
   // Called-back when the mouse-pointer has been moved onto the source component but no buttons have been pushed.
```

```
public void keyTyped(KeyEvent e)
   // Called-back when a key has been typed (pressed and released).
public void keyPressed(KeyEvent e)
public void keyReleased(KeyEvent e)
   // Called-back when a key has been pressed or released.
```

Using the Same Listener Instance for All the Buttons

- (Besides the getActionCommand(), which is only available for ActionEvent, you can) use the getSource() method, which is available to all event objects, to retrieve a reference to the source object that has fired the event. getSource() returns a java.lang.Object. You may need to downcast it to the proper type of the source object.

```
btnCountUp = new Button("Count Up");
add(btnCountUp);
// Construct an anonymous instance of an anonymous inner class.
// The source Button adds the anonymous instance as ActionEvent listener
btnCountUp.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent evt) {
      ++count;
      tfCount.setText(count + "");
   }
});

btnCountDown = new Button("Count Down");
add(btnCountDown);
btnCountDown.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent evt) {
      count--;
      tfCount.setText(count + "");
   }
});

btnReset = new Button("Reset");
add(btnReset);
btnReset.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent evt) {
      count = 0;
      tfCount.setText("0");
   }
});
```

Anonymous Inner Class for Each Source

```
25        // Allocate an instance of inner class BtnListener.
26        AllButtonsListener listener = new AllButtonsListener();
27        // Use the same listener instance to all the 3 Buttons.
28        btnCountUp.addActionListener(listener);
29        btnCountDown.addActionListener(listener);
30        btnReset.addActionListener(listener);
31
32        setTitle("AWT Counter");
33        setSize(400, 100);
34        setVisible(true);
35     }
36
37     // The entry main method
38     public static void main(String[] args) {
39        new AWTCounter3ButtonsGetSource();  // Let the constructor do the job
40     }
41
42     /**
43      * AllButtonsListener is a named inner class used as ActionEvent listener for all the Buttons.
44      */
45     private class AllButtonsListener implements ActionListener {
46        @Override
47        public void actionPerformed(ActionEvent evt) {
48           // Need to determine which button has fired the event.
49           Button source = (Button)evt.getSource();
50                 // Get a reference of the source that has fired the event.
51                 // getSource() returns a java.lang.Object. Downcast back to Button.
52           if (source == btnCountUp) {
53              ++count;
54           } else if (source == btnCountDown) {
55              --count;
56           } else {
57              count = 0;
58           }
59           tfCount.setText(count + "");
60        }
61     }
```

Using the Same Listener Instance for All the Buttons

Event Listener's Adapter Classes

- a WindowEvent listener is required to implement the WindowListener interface, which declares 7 abstract methods, although we are only interested in windowClosing().
- An adapter class called WindowAdapter is therefore provided, which implements the WindowListener interface and provides default implementations to all the 7 abstract methods. You can then derive a subclass from WindowAdapter and override only methods of interest and leave the rest to their default implementation.
- Similarly, adapter classes such as MouseAdapter, MouseMotionAdapter, KeyAdapter, FocusAdapter are available for MouseListener, MouseMotionListener, KeyListener, and FocusListener, respectively.
- There is no ActionAdapter for ActionListener, because there is only one abstract method (i.e. actionPerformed()) declared in the ActionListener interface. This method has to be overridden and there is no need for an adapter.

Layout Managers and Panel

- AWT provides the following layout managers (in package java.awt): FlowLayout, GridLayout, BorderLayout, GridBagLayout, BoxLayout, CardLayout, and others.
- Swing added more layout manager in package javax.swing.

In the `java.awt.FlowLayout`, components are arranged from left-to-right inside the container in the order that they are added (via method `aContainer.add(aComponent)`). When one row is filled, a new row will be started. The actual appearance depends on the width of the display window.

Constructors

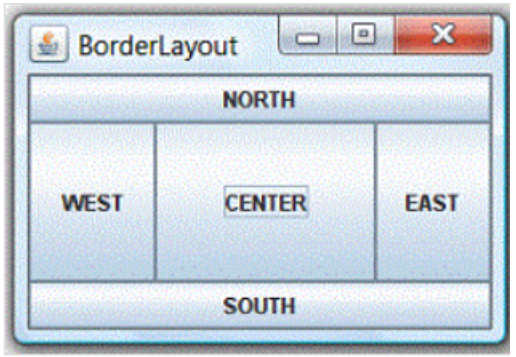
```
public FlowLayout();
public FlowLayout(int alignment);
public FlowLayout(int alignment, int hgap, int vgap);
  // alignment: FlowLayout.LEFT (or LEADING), FlowLayout.RIGHT (or TRAILING), or FlowLayout.CENTER
  // hgap, vgap: horizontal/vertical gap between the components
  // By default: hgap = 5, vgap = 5, alignment = FlowLayout.CENTER
```

In `java.awt.GridLayout`, components are arranged in a grid (matrix) of rows and columns inside the `Container`. Components are added in a left-to-right, top-to-bottom manner in the order they are added (via method *`aContainer`*`.add(`*`aComponent`*`)`).

Constructors

```
public GridLayout(int rows, int columns);
public GridLayout(int rows, int columns, int hgap, int vgap);
      // By default: rows = 1, cols = 0, hgap = 0, vgap = 0
```

In `java.awt.BorderLayout`, the container is divided into 5 zones: `EAST`, `WEST`, `SOUTH`, `NORTH`, and `CENTER`. Components are added using method *`aContainer`*`.add(`*`aComponent`*`,` *`zone`*`)`, where *`zone`* is either `BorderLayout.NORTH` (or `PAGE_START`), `BorderLayout.SOUTH` (or `PAGE_END`), `BorderLayout.WEST` (or `LINE_START`), `BorderLayout.EAST` (or `LINE_END`), or `BorderLayout.CENTER`.



You need not place components to all the 5 zones. The `NORTH` and `SOUTH` components may be stretched horizontally; the `EAST` and `WEST` components may be stretched vertically; the `CENTER` component may stretch both horizontally and vertically to fill any space left over.

Constructors

```
public BorderLayout();
public BorderLayout(int hgap, int vgap);
      // By default hgap = 0, vgap = 0
```

ContentPane / Panel

- The JFrame's method getContentPane() returns the content-pane (which is a java.awt.Containter) of the JFrame. You can then set its layout (the default layout is BorderLayout), and add components into it.

```
Container cp = getContentPane();
```

- An AWT Panel is a rectangular pane, which can be used as sub-container to organized a group of related components in a specific layout (e.g., FlowLayout, BorderLayout). Panels are secondary containers, which shall be added into a top-level container (such as Frame), or another Panel.

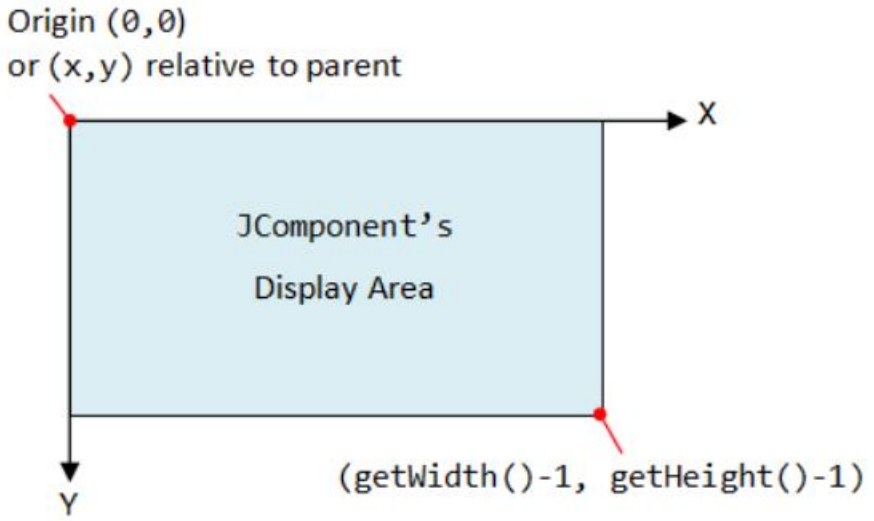
The java.awt.Graphics Class:
Graphics Context and Custom Painting
https://www3.ntu.edu.sg/home/ehchua/programming/java/J4b_CustomGraphics.html

- A graphics context provides the capabilities of drawing on the screen. The graphics context maintains states such as the color and font used in drawing, as well as interacting with the underlying operating system to perform the drawing. In Java, custom painting is done via the java.awt.Graphics class, which manages a graphics context, and provides a set of device-independent methods for drawing texts, figures and images on the screen on different platforms.
- The Graphics class provides methods for drawing three types of graphical objects: Text Strings, Vector-graphic primitives and Shapes, and Bitmap images.

```
// Drawing (or printing) texts on the graphics screen:
drawString(String str, int xBaselineLeft, int yBaselineLeft);

// Drawing lines:
drawLine(int x1, int y1, int x2, int y2);
drawPolyline(int[] xPoints, int[] yPoints, int numPoint);

// Drawing primitive shapes:
drawRect(int xTopLeft, int yTopLeft, int width, int height);
 drawOval(int xTopLeft, int yTopLeft, int width, int height);
drawArc(int xTopLeft, int yTopLeft, int width, int height, int startAngle, int arcAngle);
draw3DRect(int xTopLeft, int, yTopLeft, int width, int height, boolean raised);
drawRoundRect(int xTopLeft, int yTopLeft, int width, int height, int arcWidth, int arcHeight)
drawPolygon(int[] xPoints, int[] yPoints, int numPoint);

// Filling primitive shapes:
fillRect(int xTopLeft, int yTopLeft, int width, int height);
fillOval(int xTopLeft, int yTopLeft, int width, int height);
fillArc(int xTopLeft, int yTopLeft, int width, int height, int startAngle, int arcAngle);
fill3DRect(int xTopLeft, int, yTopLeft, int width, int height, boolean raised);
fillRoundRect(int xTopLeft, int yTopLeft, int width, int height, int arcWidth, int arcHeight)
fillPolygon(int[] xPoints, int[] yPoints, int numPoint);

// Drawing (or Displaying) images:
drawImage(Image img, int xTopLeft, int yTopLeft, ImageObserver obs); // draw image with its size
drawImage(Image img, int xTopLeft, int yTopLeft, int width, int height, ImageObserver o); // resize image on screen
```

Graphics Coordinate System

- In Java Windowing Subsystem (like most of the 2D Graphics systems), the origin (0,0) is located at the top-left corner.
- EACH component/container has its own coordinate system, ranging for (0,0) to (width-1, height-1) as illustrated.
- You can use method getWidth() and getHeight() to retrieve the width and height of a component/container. You can use getX() or getY() to get the top-left corner (x,y) of this component's origin relative to its parent.

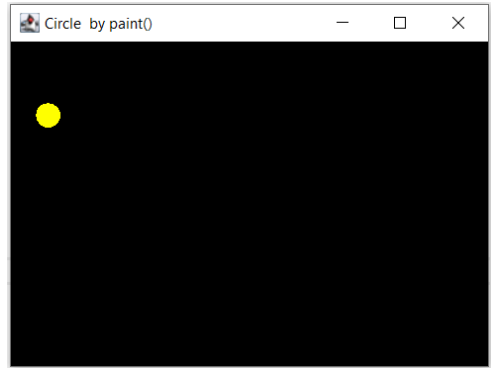


Custom Painting

- Under Swing, custom painting is usually performed by extending (i.e., subclassing) a JPanel as the drawing canvas and override the paintComponent(Graphics g) method to perform your own drawing with the drawing methods provided by the Graphics class.
- Refreshing the Display via repaint()
- At times, we need to explicitly refresh the display (e.g., in game and animation). We shall NOT invoke paintComponent(Graphics) directly. Instead, we invoke the JComponent's repaint() method. The Windowing Subsystem will in turn call back the paintComponent() with the current Graphics context and execute it in the event-dispatching thread for thread safety. You can repaint() a particular JComponent (such as a JPanel) or the entire JFrame. The children contained within the JComponent will also be repainted.

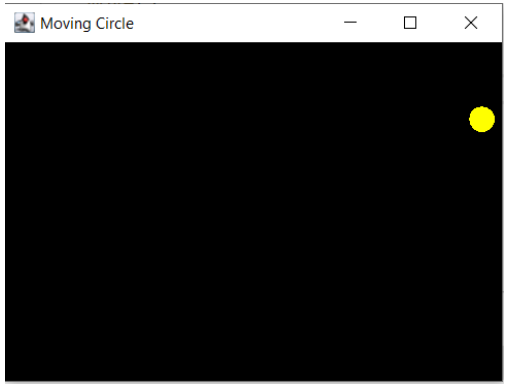
```java
import javax.swing.*;
import java.awt.*;
public class Graphic1 extends JFrame {
  static private int width = 400;
  static private int height = 300;
  Graphic1() {
    setSize(width, height);
    int xPos = 20;       int yPos = 50;
    int radius = 10;
    //Container cp = getContentPane();
    add(new JPanel() {
        //JPanel x = new JPanel() {
      //@Override
      public void paintComponent(Graphics g) {
        super.paintComponent(g);
        setBackground(Color.BLACK);
        g.setColor(Color.YELLOW);
        g.fillOval(xPos,yPos,radius * 2, radius * 2);
      }
    } ) ;
    setTitle("Circle");
    setVisible(true);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
  public static void main(String [] args) {
   Graphic1 g = new Graphic1();
  }
}
```

```java
import javax.swing.*;
import java.awt.*;
public class Graphic3 extends JFrame {
  static private int width = 400;
  static private int height = 300;
  Graphic3() {
    setSize(width, height);
    int xPos = 20; int yPos = 50;
    int radius = 10;

    add(new JPanel() {

      @Override
      public void paint(Graphics g) {
        super.paint(g);
        setBackground(Color.BLACK);
        g.setColor(Color.YELLOW);
        g.fillOval(xPos,yPos,radius * 2, radius * 2);
      }
    } ) ;
    setTitle("Circle By paint()");
    setVisible(true);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
  public static void main(String [] args) {
        Graphic3 g = new Graphic3();
  }
}
```



paint()

- Painting actually begins higher up the class hierarchy, with the (java.awt.Component) paint() method
  - javax.swing.JComponent extends this class and further factors the paint method into three separate methods, which are invoked in the following order:
    - protected void paintComponent(Graphics g)
    - protected void paintBorder(Graphics g)
    - protected void paintChildren(Graphics g)
- The painting for the standard Swing components proceeds as follows.
  - paint() invokes paintComponent().
  - If the ui property is non-null, paintComponent() invokes ui.update().
  - If the component's opaque property is true, ui.update() fills the component's background with the background color and invokes ui.paint().
  - ui.paint() renders the content of the component.




```
public class Graphic4 extends JFrame {
  static private int width = 400;
  static private int height = 300;
  int xPos = 20;  int yPos = 50;
  int radius = 10;
  Graphic4() {
    setSize(width, height);
    add(new JPanel() {
        @Override
        public void paint(Graphics g) {
          super.paint(g); //super.paintComponent(g);
          setBackground(Color.BLACK);
          g.setColor(Color.YELLOW);
          g.fillOval(xPos,yPos,radius * 2,radius * 2);
          move();
        }
        private void move() {
          if (xPos < getWidth() - 20 - 6) {
              xPos += 10;
              repaint();
        }  }
      } ) ;
      setTitle("Moving Circle");
      setVisible(true);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
  }
  public static void main(String [] args) {Graphic4 g = new Graphic3();}
}
```

java.awt.Color

```
RED        : java.awt.Color[r=255, g=0,   b=0]
GREEN      : java.awt.Color[r=0,   g=255, b=0]
BLUE       : java.awt.Color[r=0,   g=0,   b=255]
YELLOW     : java.awt.Color[r=255, g=255, b=0]
MAGENTA    : java.awt.Color[r=255, g=0,   b=255]
CYAN       : java.awt.Color[r=0,   g=255, b=255]
WHITE      : java.awt.Color[r=255, g=255, b=255]
BLACK      : java.awt.Color[r=0,   g=0,   b=0]
GRAY       : java.awt.Color[r=128, g=128, b=128]
LIGHT_GRAY: java.awt.Color[r=192, g=192, b=192]
DARK_GRAY : java.awt.Color[r=64,  g=64,  b=64]
PINK       : java.awt.Color[r=255, g=175, b=175]
ORANGE     : java.awt.Color[r=255, g=200, b=0]
```

You can also use the RGB values or RGBA value (A for alpha to specify transparency/opaque) to construct your own color via constructors:

```
Color(int r, int g, int b);             // between 0 and 255
Color(float r, float g, float b);       // between 0.0f and 1.0f
Color(int r, int g, int b, int alpha);          // between 0 and 255
Color(float r, float g, float b, float alpha); // between 0.0f and 1.0f
   // alpha of 0 for totally transparent, 255 (or 1.0f) for totally opaque
   // The default alpha is 255 (or 1.0f) for totally opaque
```

java.awt.Font

- The class java.awt.Font represents a specific font face, which can be used for rendering texts. You can use the following constructor to construct a Font instance:

```
public Font(String name, int style, int size);
// name:  Family name "Dialog", "DialogInput", "Monospaced", "Serif", or "SansSerif" or
//        Physical font found in this GraphicsEnvironment.
//        You can also use String constants Font.DIALOG, Font.DIALOG_INPUT, Font.MONOSPACED,
//          Font.SERIF, Font.SANS_SERIF (JDK 1.6)
// style: Font.PLAIN, Font.BOLD, Font.ITALIC or Font.BOLD|Font.ITALIC (Bit-OR)
// size:  the point size of the font (in pt) (1 inch has 72 pt).
```

Graphics Class' drawImage()

- The drawImage() method requires an Image instance, which can be obtained via ImageIcon's getImage() method; or via static method ImageIO.read().
- javax.swing.ImageIcon
  - The javax.swing.ImageIcon class represents an icon, which is a fixed-size picture, typically small-size and used to decorate components.

KeyPoints

- Java swing components
  - Frame → (Container) →Add components → implement actionListener()
- Java awt.events
  - Register (addActionListener())
  - actionPerform() //Anonymous or Lambda
- Graphics g
  - .paint() method