Unit-1

GUI Programming

Asst. Prof. Roshan Tandukar

Graphical User Interface

- An application that uses graphical objects to interact with users
- Different Java APIs for graphics programming are available
- Most commonly used are:

1. AWT(Abstract Windowing Toolkit)
2. Swing
3. JavaFX

Java APIs

1. AWT (Abstract Windowing Toolkit)

- AWT API was introduced in JDK 1.0.
- Most of the AWT components have become obsolete and should be replaced by newer Swing components.

2. Swing

- Swing API is a much more comprehensive set of graphics libraries that enhances the AWT
- Introduced as part of Java Foundation Classes (JFC) after the release of JDK 1.1.

3. JavaFX

- In 2008, a new Java GUI toolkit was released. It was created in order to address new demands in graphical computing such as advanced animations and multitouch support.
- JavaFX is a software platform for developing and delivering rich internet applications (RIAs) that can run across a wide variety of devices
- It is fully integrated with recent versions of Java SE Runtime Environment (JRE) and the Java Development Kit (JDK).

AWT (Abstract Windowing Toolkit)

- A set of application program interfaces (APIs) used in Java to create graphical user interface (GUI) objects, such as labels, buttons, scroll bars, and window-based applications in java.
- Its Java's original windowing, graphics, and user-interface widget toolkit.
  - Components are platform-dependent i.e. components are displayed according to the view of operating system.
  - It is heavyweight i.e. its components are using the resources of OS.
- Package: java.awt
- Example of classes:

  Frame, TextField, Label, TextArea, Button, RadioButton, CheckBox, Choice, List etc.

AWT Packages

- AWT is huge! But Swing is even bigger, with more packages.
- Fortunately, only 2 packages - java.awt and java.awt.event - are commonly-used.
- The java.awt package contains the core AWT graphics classes:
  - GUI Component classes (such as Button, TextField, and Label),
  - GUI Container classes (such as Frame, Panel, Dialog and ScrollPane),
  - Layout managers (such as FlowLayout, BorderLayout and GridLayout),
  - Custom graphics classes (such as Graphics, Color and Font).
- The java.awt.event package supports event handling:
  - Event classes (such as ActionEvent, MouseEvent, KeyEvent and WindowEvent),
  - Event Listener Interfaces (such as ActionListener, MouseListener, KeyListener and WindowListener),
  - Event Listener Adapter classes (such as MouseAdapter, KeyAdapter, and WindowAdapter).

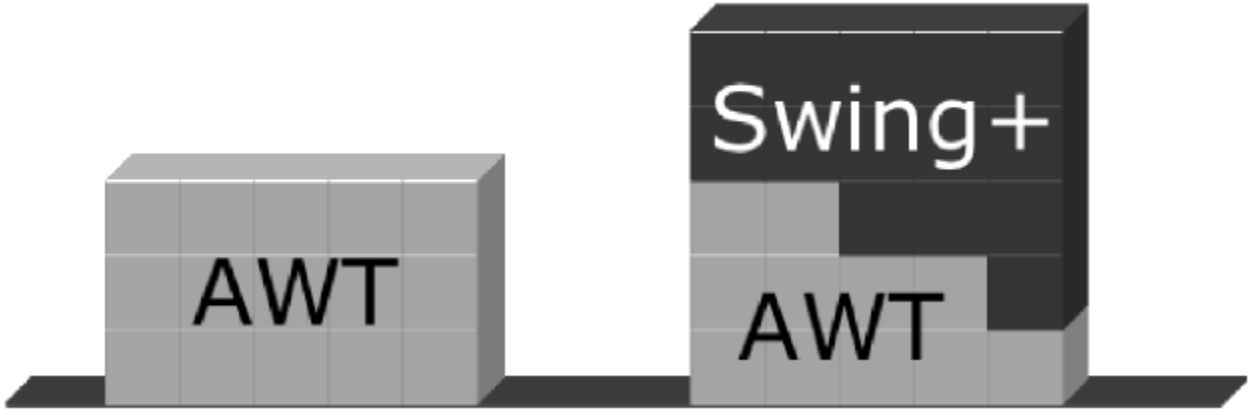
Swing

- The primary Java GUI widget toolkit.
- A part of Oracle's Java Foundation Classes (JFC) — an API for providing a graphical user interface (GUI) for Java programs.
- More sophisticated set of GUI components than the earlier Abstract Window Toolkit (AWT).
    - Provides **a native look and feel** that emulates the look and feel of several platforms, and also supports **a pluggable look and feel**
    - Unlike AWT components, Swing components are not implemented by platform specific code. Instead they are written entirely in Java and therefore are **platform-independent**.
- Package: javax.swing
- Example of classes: JFrame, Jbutton,, Jlabel, JTextField, JToolBar, JRadioBox etc.

Swing Features

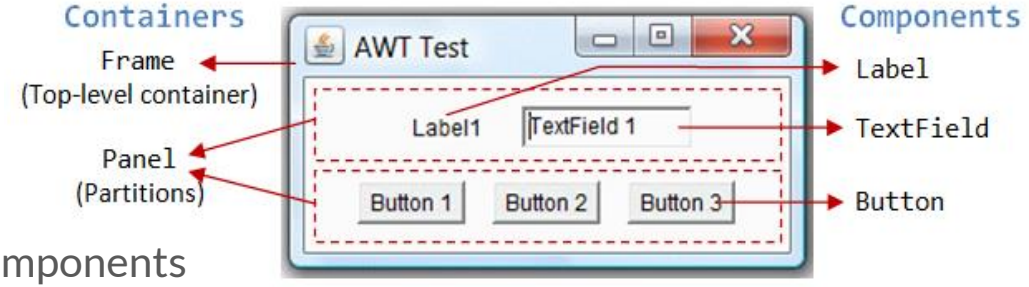
- **Light Weight**
  - Component are independent of native Operating System's API as Swing API controls are rendered mostly using pure JAVA code instead of underlying operating system calls.
- **Rich controls**
  - Provides a rich set of advanced controls like Tree, TabbedPane, slider, colourpicker, table controls
- **Highly Customizable**
  - Can be customized in very easy way as visual appearance is independent of internal representation.
- **Pluggable look-and-feel**
  - SWING based GUI Application look and feel can be changed at run time based on available values.

AWT and Swing



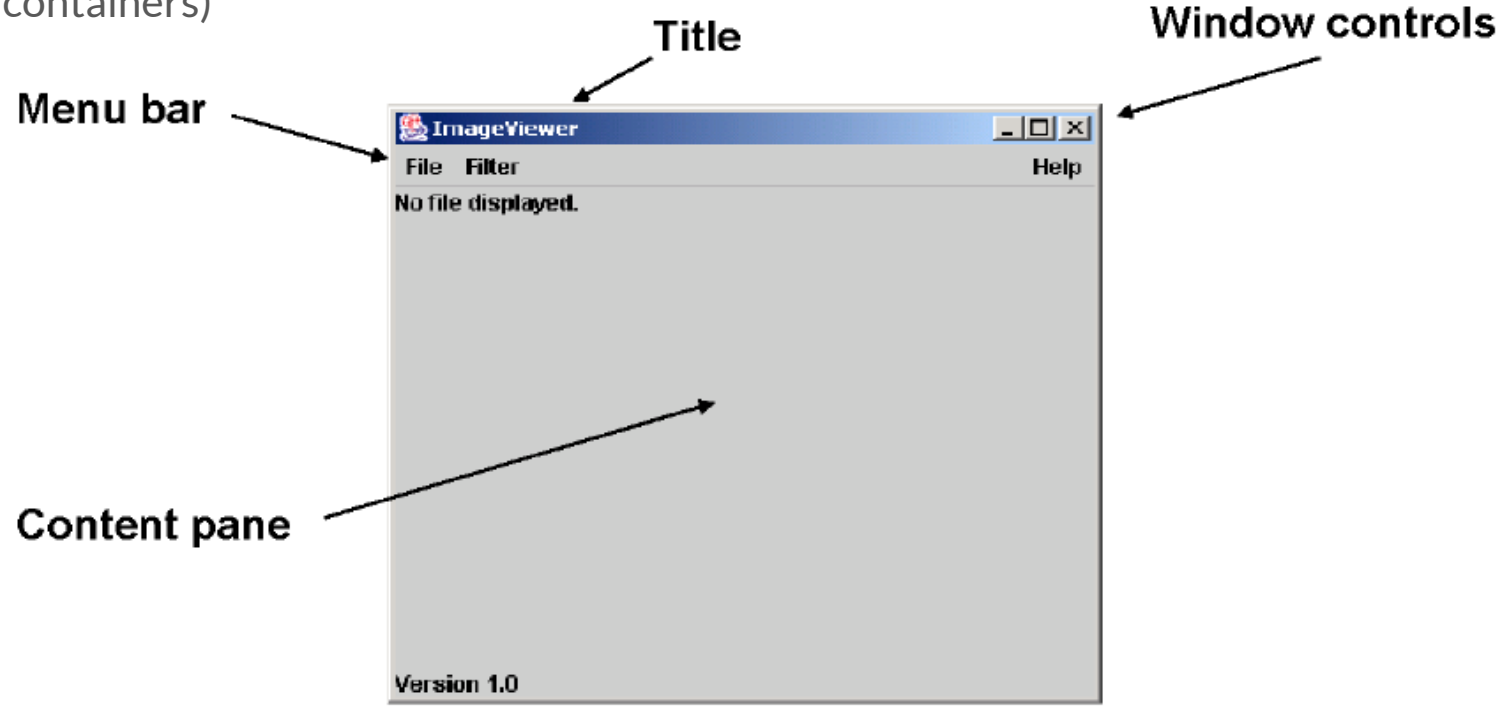
Terminology

- Window
  - A first-class component of the graphical desktop(Also called a top-level container)
  - Examples: frame, dialog box, applet
- Component
  - A GUI widget that resides in a window(Also called controls in many other languages)
  - Examples: button, text box, label
- Container
  - A component that hosts (holds) components
  - Examples: panel, box



JFrame and its Elements

- JFrame is the Swing Window
- JPanel (aka a pane) is the container to which you add your components (or other containers)



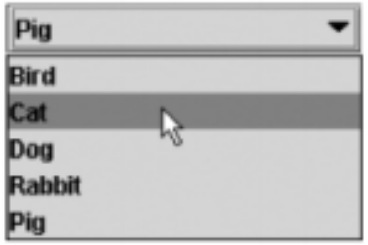
Components in a JFrame

Components

JButton

JTextField

JComboBox

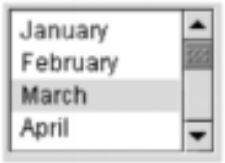


JColorChooser

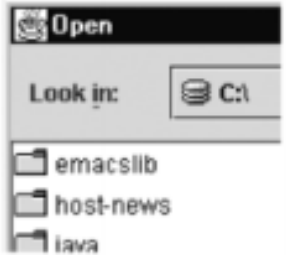
JCheckBox

JSlider

JList



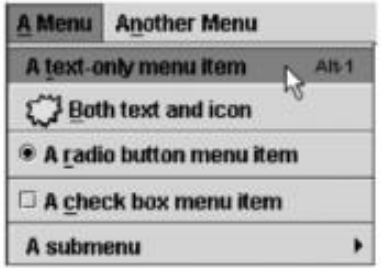
JFileChooser



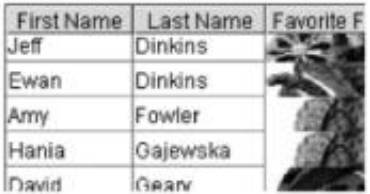
JRadioBox

JToolBar

JMenuBar, JMenu, JMenuItem

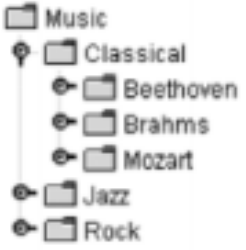


JTable



JLabel

JTree



Swing/AWT inheritance hierarchy

```
Component   (AWT)
    Window
        Frame
            JFrame   (Swing)
            JDialog
    Container
        Jcomponent  (Swing)
            JButton          JColorChooser     JFileChooser
            JComboBox        JLabel            JList
            JMenuBar         JOptionPane       JPanel
            JPopupMenu       JProgressBar      JScrollbar
            JScrollPane      JSlider           JSpinner
            JSplitPane       JTabbedPane       JTable
            JToolbar         JTree             JTextArea
            JTextField       ...
```

To create a simple Swing application

- Make a Window (a JFrame)
- Make a container (a JPanel)
  - Put it in the window
- Add components (Buttons, Boxes, etc.) to the container
  - Use layouts to control positioning
  - Set up observers (a.k.a. listeners) to respond to events
  - Optionally, write custom widgets with application-specific display logic
- Set up the window to display the container
- Then wait for events to arrive

Example

```
import java.awt.Container;
import javax.swing.JFrame;
import javax.swing.JLabel;
class App{
        public static void main(String args[]){
                JLabel label=new JLabel("This is a label...");                    // Creating a label
                Container contentPane=frame.getContentPane();  // Creating a container, inserting it to the frame
                 contentPane.add(label);                                          //Adding label to the container

                 JFrame frame=new JFrame("This is first application");            //Creating a frame
                frame.setSize(300,300);                                           //Defining size of the frame
                frame.setVisible(true);                                           //Setting the visibility of frame to true
        }
}
```

Example

```
import javax.swing.*;
   public class FrameExample{
    public static void main(String[] args)
    {
              JLabel label = new JLabel("Hello World !");          // Make a JLabel;
             JPanel panel = new JPanel();                          // Make a JPanel;
              panel.add(label);                                    // Add label to panel

             JFrame frame=new JFrame("This is first application"); //Creating a frame
             frame.getContentPane().add(panel);                   // Add panel to Frame
              frame.setSize(400,300);
              frame.setVisible(true);
    }
}
```

Why to use Panel?

1.to group components together.
2.to devise complex interfaces, as each **panel** can have a different layout
3.to build reusable components and isolate responsibility.

Components

- JButton

```
JButton button = new JButton(“Add”);
button.setMnemonic(‘A’);
button.setToolTipText(“Add a record”);
```

- JFrame

```
JFrame frame = new  JFrame("A Title");
```

- JTextField

```
JTextField data = new JTextField(“”)
String text = data.getText();
```

- JLabel

```
JLabel label = new JLabel(“label  to display”);
label.setText(“new Text”);
```

- JComboBox

```
JComboBox box = new JComboBox(array or object);
```

- JRadioButton

```
JRadioButton button= new JRadioButton(“end”, true);
```

- JCheckBox

```
JCheckBox box = new JCheckBox((“Bold”, true);
```

- Jlist

```
Jlist list = new Jlist(array or  object);
```

- JScrollPane

```
JScrollPane scroll = new JScrollPane(array or object);
```

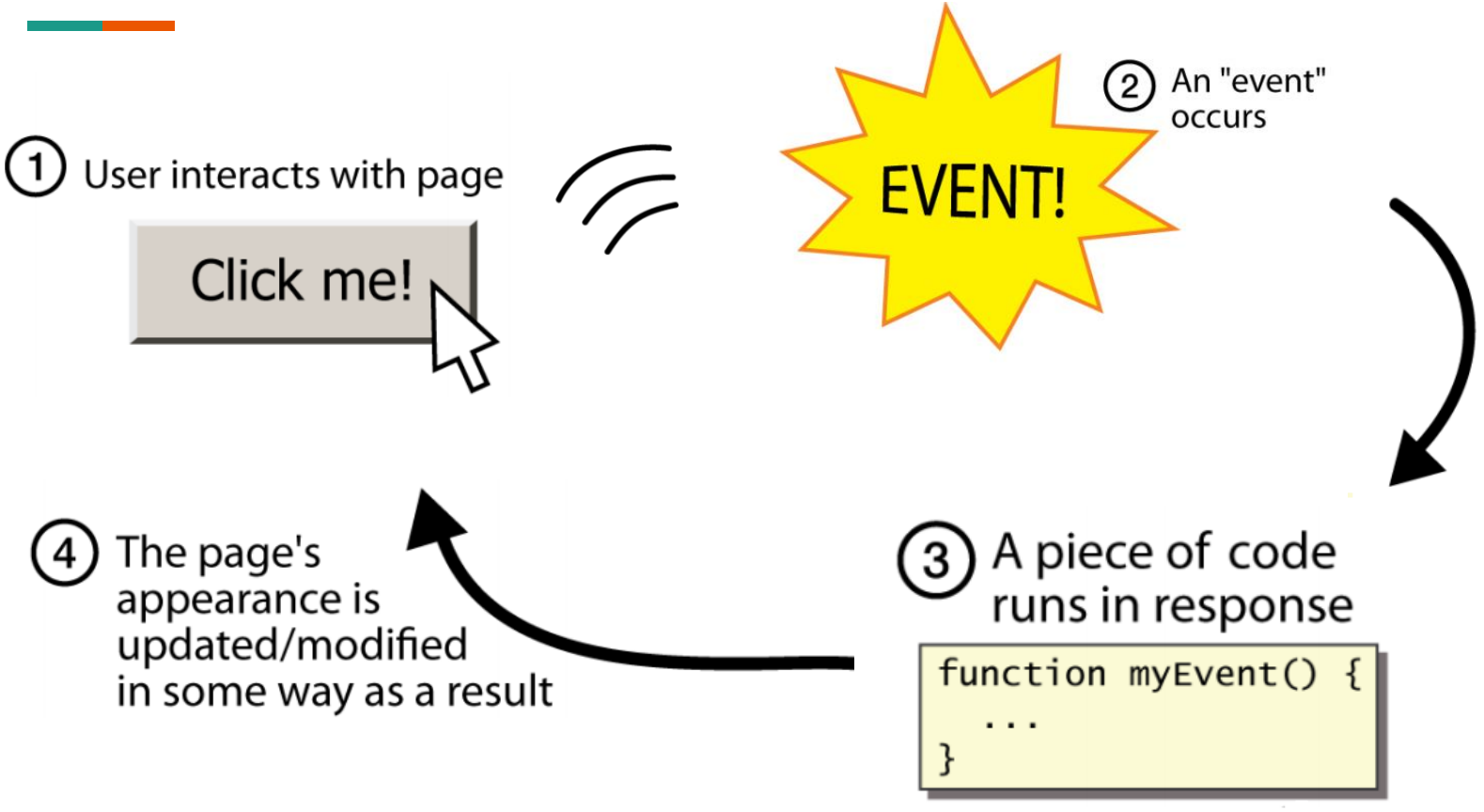
- JtabbedPane

```
JTabbedPane pane = new JTabbedPane();
pane.addTab(“label”, container);
```

Events and Event handling

- An **event** can be defined as a type of signal to the program that something has happened.
- An **event in Java** is an object that is created when something changes within a graphical user interface.
- The event is generated by external user actions such as mouse movements, mouse button clicks, and keystrokes, or by the operating system, such as a timer.
- If a user clicks on a button, clicks on a combo box, or types characters into a text field, etc., then an event triggers, creating the relevant event object.
- This behavior is part of Java's Event Handling mechanism

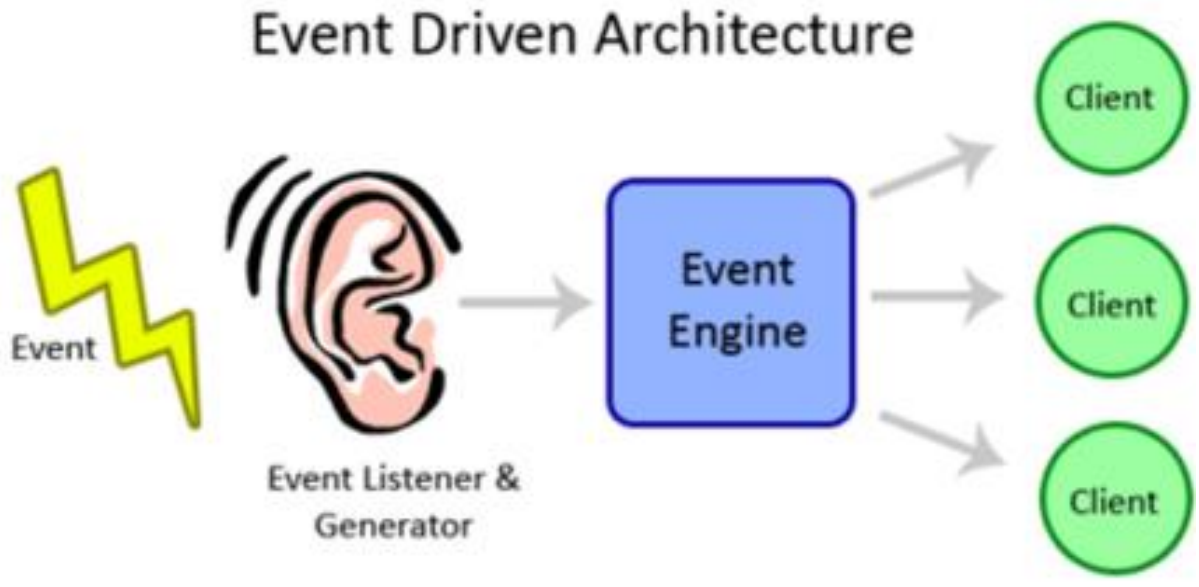
Event Handling



Event Handling Model

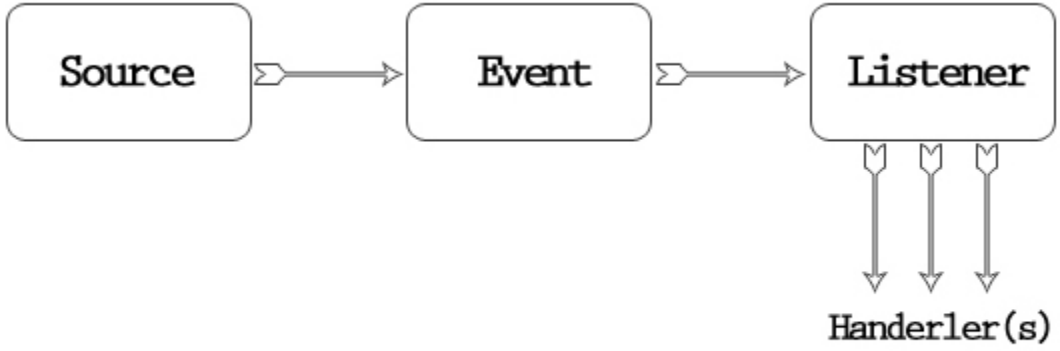
- In the event model, there are three participants:

1. Event source
2. Event object
3. Event listener



Event Handling Model

- Event source is the object whose state changes and generates Events. (i.e. an object that is created when an event occurs.)
- Event object (Event) encapsulates the state changes in the event source.
- The event listener, the object that "listens" for events and processes them when they occur.
- Event source object delegates the task of handling an event to the event listener.



Event-Driving Programming Model

Button

- Class: Jbutton
- Package: javax.swing.Jbutton;
- Constructor: new Jbutton(String text)

GUI Codin Part:

- Import javax.swing.Jbutton; //import JButton
- JButton button; //Declare JButton
- Button= new Jbutton("Press me!"); //Initialize JButton
- ContentPane.add(button); //Add button to the container

Adding ActionListener:

- button.addActionListerner(this); //Add action listener

Handling action event:

- Using getActionComman();
- Using getSource();

Text Field

- Class: JTextField
- Package: javax.swing.JTextfield;
- Constructor: new JTextField() or new JTextField(int columns)

GUI Coding Part:

- Import javax.swing.JTextfield; //import JTextfield
- JTextfield text; //Declare JTextfield
- text= new JTextfield (); //Initialize JTextfield
- ContentPane.add(text); //Add Textfield to the container

Text Field

- Class: JTextField
- Package: javax.swing.JTextfield;
- Constructor: new JTextField() or new JTextField(int columns)

Getting value from Textfield

- getText() is used.
- String input=text.getText();

Parsing to numbers

- Int num=Integer.parseInt(input);
- Double dnum=Double.parseDouble(input);

Events

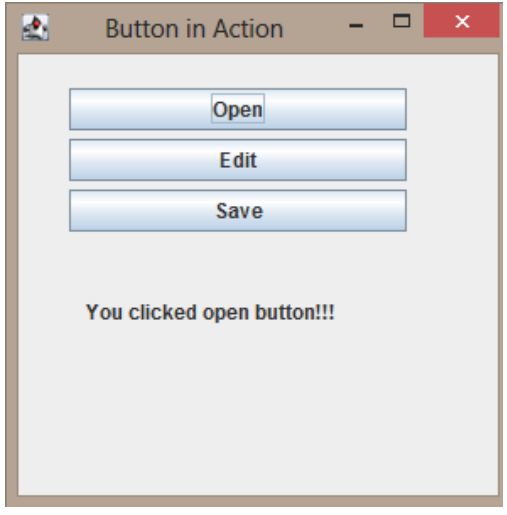
User Action	Source Object	Event Type Generated
Click a button	Jbutton	ActionEvent
Click a check box	JCheckBox	ItemEvent, ActionEvent
Click a radio button	JRadioButton	ItemEvent, ActionEvent
Press return on a text field	JTextField	ActionEvent
Select a new item	JComboBox	ItemEvent, ActionEvent
Select an item from a List	JList	ListSelectionEvent
Window opened, closed, etc.	Window	WindowEvent
Mouse pressed, released, etc.	Any Component	MouseEvent
Key released, pressed, etc.	Any Component	KeyEvent

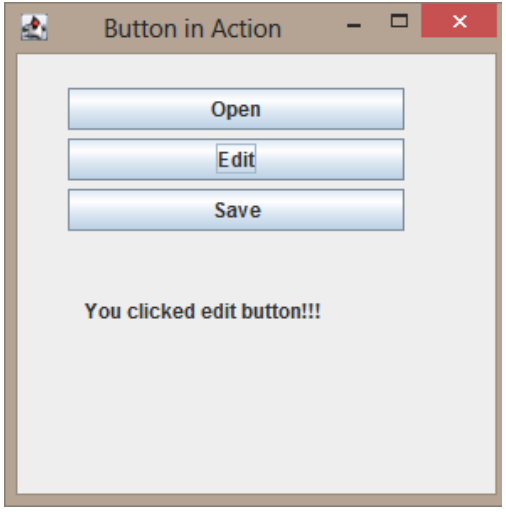
Events and Event Listener

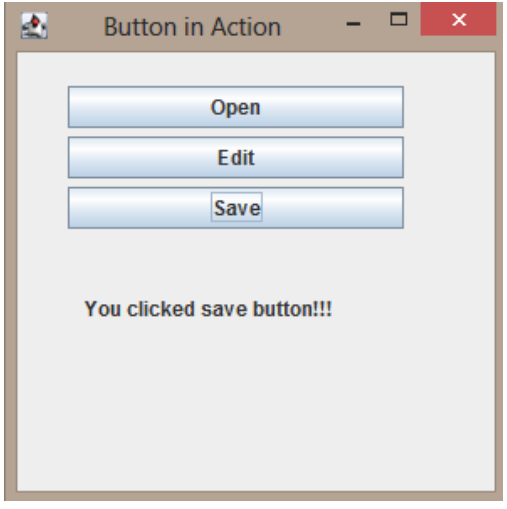
Events	Description	Related listener
ActionEvent	Represents a graphical element is clicked, such as a button or item in a list.	ActionListener
ContainerEvent	Represents an event that occurs to the GUI's container itself, for example, if a user adds or removes an object from the interface.	ContainerListener
KeyEvent	Represents an event in which the user presses, types or releases a key	KeyListener
WindowEvent	Represents an event relating to a window, for example, when a window is closed, activated or deactivated	WindowListener
MouseEvent	Represents any event related to a mouse, such as when a mouse is clicked or pressed.	MouseListener

Example

- Create a form with three buttons which displays which button is pressed in a label.







Code

```
import java.awt.*;
import javax.swing.*;
 public class EventSample implements ActionListener{
        JButton open, edit, save;              // three Button reference variables
        JLabel label;
        public void setButtons(){
                open = new JButton("Open");
                open.setBounds(30,20,200,25);
                edit = new JButton("Edit");
                edit.setBounds(30,50,200,25);
                save = new JButton("Save");
                save.setBounds(30,80,200,25);

                open.addActionListener(this); // link the Java button with the ActionListener
                edit.addActionListener(this);
                save.addActionListener(this);

                label = new JLabel();
                label.setBounds(40,140,150,25);

JFrame frame=new JFrame();
JPanel panel=new JPanel();
panel.setLayout(null);

panel.add(open);
panel.add(edit);
panel.add(save);
panel.add(label);

frame.add(panel);
frame.setTitle("Button in Action");
frame.setSize(300, 300);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_
CLOSE);
frame.setVisible(true);
                                                                }
```

Code(Contd.)

actionPerformed is a method required for all ActionListeners

```
public void actionPerformed(ActionEvent e){
        if e.getSource().equals(open)){          //if(e.getActionCommand().equals("Open"))
                label.setText("You clicked open button!!!");
        }
       else if(e.getSource().equals(edit)){
                label.setText("You clicked edit button!!!");
        }
        else if(e.getSource().equals(save) {
                label.setText("You clicked save button!!!");
         }
   }
   public static void main(String args[]){
     EventSample sam=new EventSample();
     sam.setButtons();
   }
}
```

getActionCommand() returns String representing the action command, set through the setActionCommand()

getSource() method returns a reference to the Component object that generated the event.

Listeners: Methods responding to Events

Examples

- **MouseListener** – respond to user mouse events
  - Add "implements MouseListener" to the GUI class
  - Code listener methods (e.g. mouseClicked()) and attach to the GUI object
- **MouseMotionListener** – respond to mouse movements
  - Add "implements MouseMotionListener" to the GUI class
  - Code listener methods (e.g. mouseMoved()) and attach to the GUI object
- **ActionListener** – Recponds once to button selections
  - Add "implements ActionListener" to the GUI class
  - Code the "actionPerformed" method and attach to the GUI object
- **ItemListener** – Responds multiple times to changes to a component
  - Add "implements ItemListener" to the GUI class
  - Code the "itemStateChanged" method
  - Attach the ItemListener to the GUI object
- **Window Listener** – respond to clicks of a frame's X button
  - Create a class that extends WindowAdapter
  - Code the WindowListener methods and attach to the frame

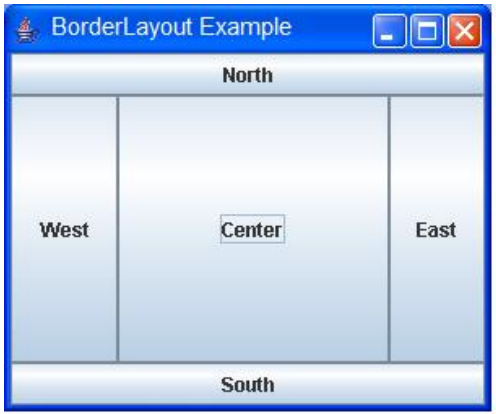
Containers - Layout

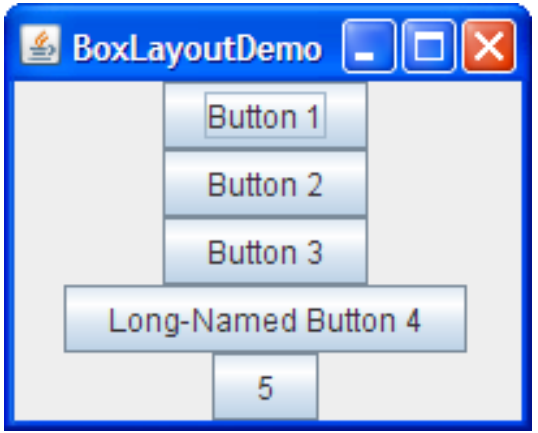
- Each container has a layout manager
  - Determines the size, location of contained widgets.
- The LayoutManagers are used to arrange components in a particular manner and determines the size and position of the components within a container
- All layout managers implement one of two interfaces defined in the java.awt package: LayoutManager or its subclass, LayoutManager2.
- LayoutManager declares a set of methods that are intended to provide a straightforward, organized means of managing component positions and sizes in a container.
- Each implementation of LayoutManager defines these methods in different ways according to its specific needs.

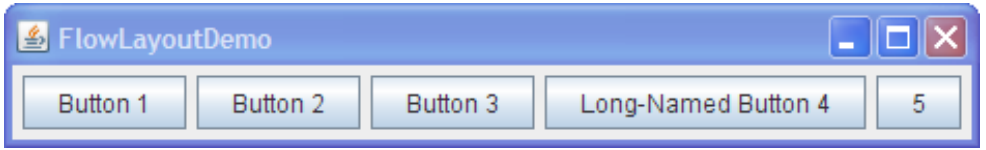
Containers - Layout

- Setting the current layout of a container:
  `void setLayout(LayoutManager lm)`
- Package: `java.awt`
- `LayoutManager` implementing classes:
  - `BorderLayout`
  - `BoxLayout`
  - `FlowLayout`
  - `GridLayout`

Containers - Layout





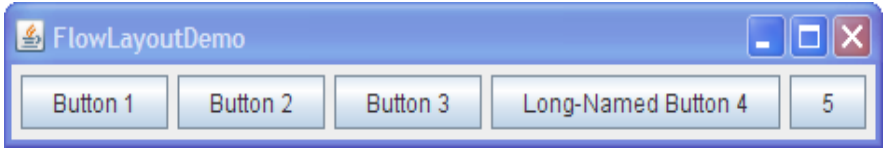


Containers - Layout

FlowLayout

class java.awt.FlowLayout

- This is a simple layout which places components from left to right in a row using the preferred component sizes (the size returned by getPreferredSize()), until no space in the container is available.
- When no space is available a new row is started.
- Because this placement depends on the current size of the container, we cannot always guarantee in advance in which row a component will be placed.
- **FlowLayout is the default layout for all JPanels** (the only exception is the content pane of a JRootPane which is always initialized with a BorderLayout).




```
panel = new JPanel(new FlowLayout());
Or,
panel.setLayout(new FlowLayout());
```

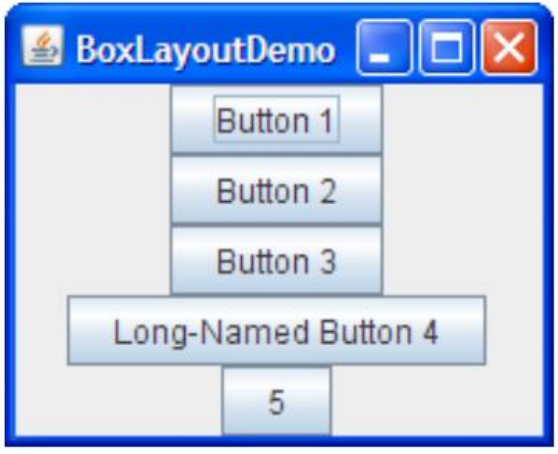
Containers - Layout

BoxLayout

class javax.swing.BoxLayout

- The BoxLayout class is used to arrange the components either vertically (along Y-axis) or horizontally (along X-axis)
- The only constructor, BoxLayout(Container target, int axis), takes a reference to the Container component it will manage and a direction (BoxLayout.X_AXIS or BoxLayout.Y_AXIS).
- Components are laid out according to their preferred sizes and they are not wrapped, even if the container does not provide enough space.

Containers - Layout




```
BoxLayout boxlayout = new BoxLayout(panel, BoxLayout.Y_AXIS);
panel.setLayout(boxlayout);
```

Containers - Layout

GridLayout

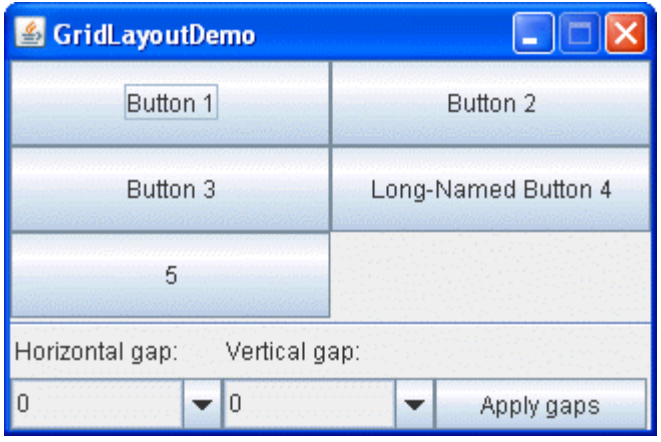
class java.awt.GridLayout

- This layout places components in a rectangular grid. There are three constructors:

1. **GridLayout()**: Creates a layout with one column per component. Only one row is used.
2. **GridLayout(int rows, int cols)**: Creates a layout with the given number of rows and columns.
3. **GridLayout(int rows, int cols, int hgap, int vgap)**: Creates a layout with the given number of rows and columns, and the given size of horizontal and vertical gaps between each row and column.

- GridLayout places components from left to right and from top to bottom, assigning the same size to each.

Containers - Layout



panel.setLayout(new GridLayout(3,2));

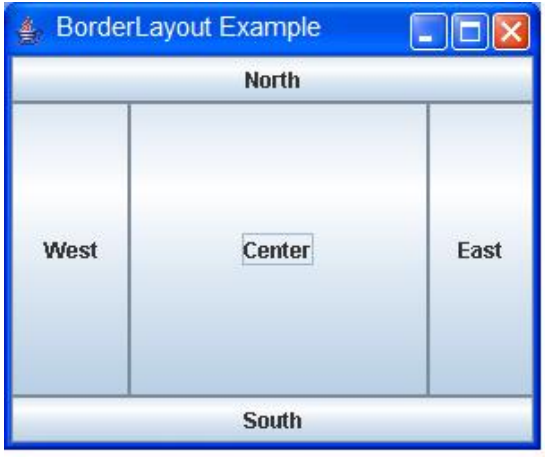
Containers - Layout

BorderLayout

class java.awt.BorderLayout

- This layout divides a container into five regions: center, north, south, east, and west.
- To specify the region in which to place a component, we use Strings of the form “Center,” “North,” and so on, or the static String fields defined in BorderLayout, which include BorderLayout.NORTH, BorderLayout.SOUTH, BorderLayout.EAST, BorderLayout.WEST, BorderLayout.CENTER.
- During the layout process, components in the north and south regions will first be allotted their preferred height (if possible) and the width of the container.
- After that, components in the east and west regions will attempt to occupy their preferred width as well as any remaining height between the north and south components.
- A component in the center region will occupy all remaining available space.
- Mostly, BorderLayout is very useful, especially in conjunction with other layouts.

Containers - Layout




```
panel.setLayout(new BorderLayout());
panel.add(btn1, BorderLayout.NORTH);
panel.add(btn2, BorderLayout.SOUTH);
panel.add(btn3, BorderLayout.EAST);
panel.add(btn4, BorderLayout.WEST);
panel.add(btn5, BorderLayout.CENTER);
```

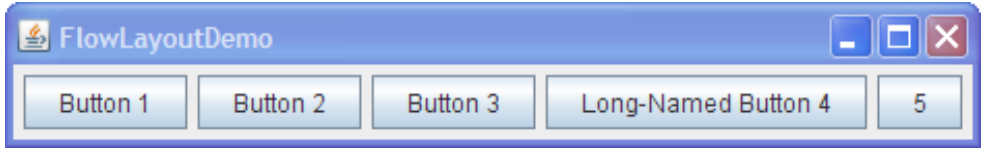
Additional Layout managers

- **CardLayout**: Unlike other layout managers, that display all the components within the container at once, a CardLayout layout manager displays only one component at a time

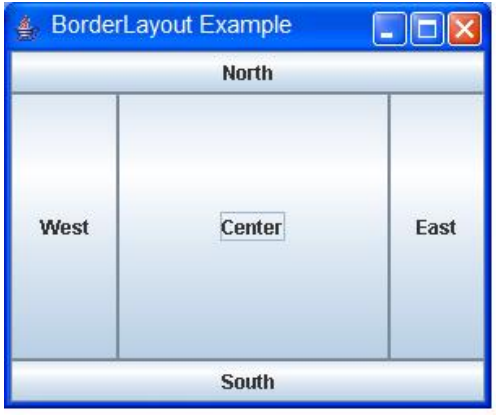
- **GridBagLayout**: Aligns components vertically, horizontally or along their baseline and each components may not be of the same size. (Each GridBagLayout object maintains a dynamic, rectangular grid of cells and each component occupies one or more cells known as its display area)

- **GroupLayout**: Groups its components and places them in a Container hierarchically and the grouping is done by instances of the Group class.

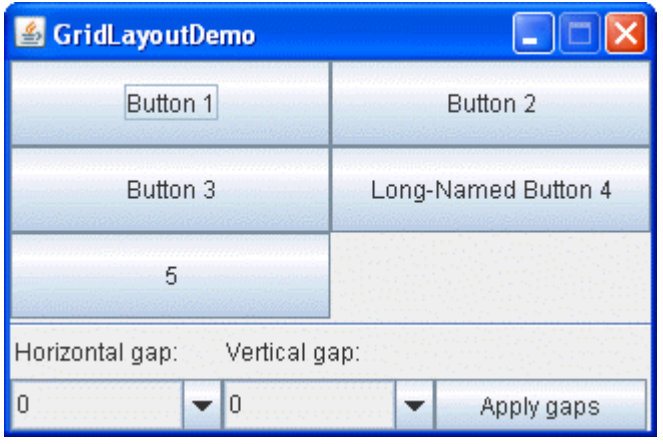
Containers - Layout




```
panel = new JPanel(new FlowLayout());
Or,
panel.setLayout(new FlowLayout());
```




```
panel.setLayout(new BorderLayout());
panel.add(btn1, BorderLayout.NORTH);
panel.add(btn2, BorderLayout.SOUTH);
panel.add(btn3, BorderLayout.EAST);
panel.add(btn4, BorderLayout.WEST);
panel.add(btn5, BorderLayout.CENTER);
```




```
panel.setLayout(new GridLayout(3,2));
```

More layout managers:

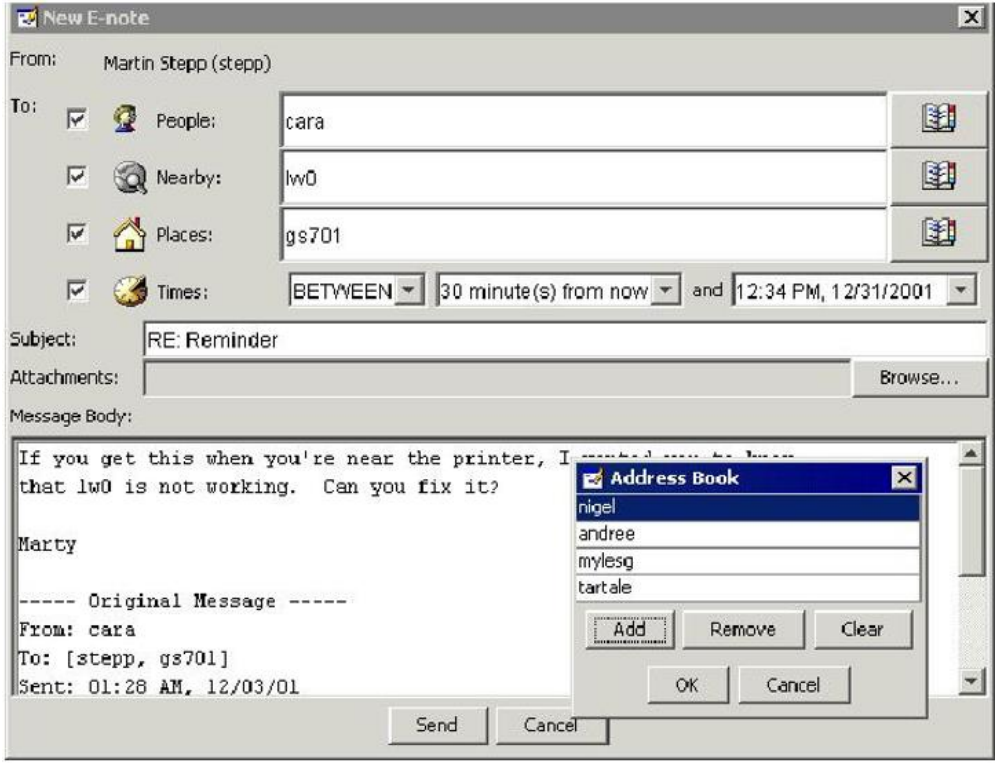
A Visual Guide to Layout Managers (The Java™ Tutorials > Creating a GUI With Swing > Laying Out Components Within a Container) (oracle.com)

Limitation in Layout

- Cannot position component as desired
- No Fixed position or exact position
- Solution for more precise location arrangement
    - Use frame.setLayout(null);
    - <object>.setBounds(x,y,width,height);
    - Eg. btn.setBounds(200,200,100,25)

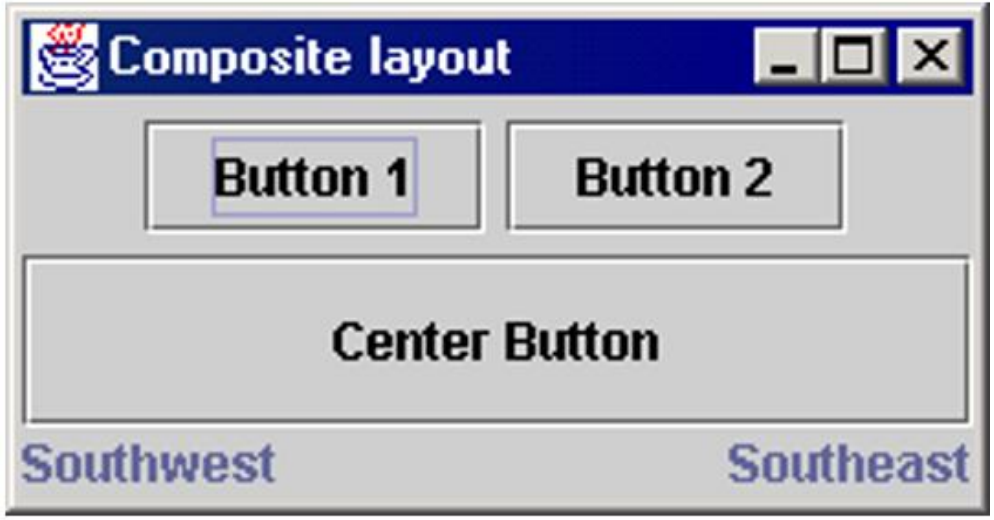
Complex layouts

- How would you create a complex window like this, using the layout managers shown?



Solution: composite layout

- create panels within panels
- each panel has a different layout, and by combining the layouts, more complex / powerful layout can be achieved
- example:
  - how many panels?
  - what layout in each?



Working with Graphics

- One of Java's initial appeals was its support for graphics that enabled programmers to visually enhance their applications.
- Java contains many more sophisticated drawing capabilities as part of the Java 2D™ API.
- Java's graphics capabilities
  - Drawing 2D shapes
  - Controlling colors
  - Controlling fonts
- Java 2D API
  - More sophisticated graphics capabilities
  - Drawing custom 2D shapes
  - Filling shapes with colors and patterns

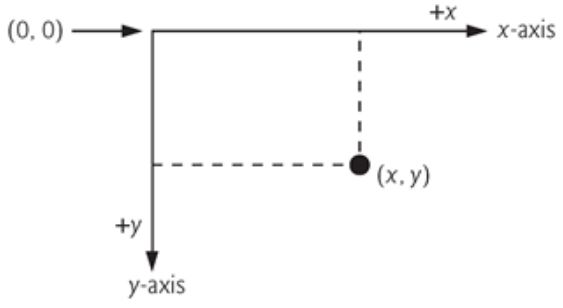
Java Classes for Graphics

- Class **Color** contains methods and constants for manipulating colors.
- Class **JComponent** contains method paintComponent, which is used to draw graphics on a component.
- Class **Font** contains methods and constants for manipulating fonts.
- Class **FontMetrics** contains methods for obtaining font information.
- Class **Graphics** contains methods for drawing strings, lines, rectangles and other shapes.
- Class **Graphics2D**, which extends class Graphics, is used for drawing with the Jav
- Class **Polygon** contains methods for creating polygons.
- Class **BasicStroke** helps specify the drawing characteristics of lines.

Graphics in Java

Coordinate system

- A scheme for identifying every point on the screen.
- The upper-left corner of a GUI component (e.g., a window) has the coordinates (0, 0).
- A coordinate pair is composed of an x-coordinate (the horizontal coordinate) and a y-coordinate (the vertical coordinate).
  - x-coordinates from left to right.
  - y-coordinates from top to bottom.



Java coordinate system. Units are measured in pixels.

Color Control

- Every color is created from a red, a green and a blue component.
- RGB values: Integers in the range from 0 to 255, or floating-point values in the range 0.0 to 1.0.
- Specifies the amount of red, the second the amount of green and the third the amount of blue.
- Larger values means more of that particular color.
- Approximately 16.7 million colors.

Color Control

- Class Color
  - Defines methods and constants for manipulating colors
  - Constants (**Color.YELLOW**)
  - Colors are also created from Red, Green and Blue components
    - RGB values (0-255)
  - JColorChooser class available in javax.swing
    - GUI component that allows user to select color
    - JColorChooser(ref to parent component, title bar string, initial selected color)

Color Control

Color constant	RGB value
`public final static Color RED`	255, 0, 0
`public final static Color GREEN`	0, 255, 0
`public final static Color BLUE`	0, 0, 255
`public final static Color ORANGE`	255, 200, 0
`public final static Color PINK`	255, 175, 175
`public final static Color CYAN`	0, 255, 255
`public final static Color MAGENTA`	255, 0, 255
`public final static Color YELLOW`	255, 255, 0
`public final static Color BLACK`	0, 0, 0
`public final static Color WHITE`	255, 255, 255
`public final static Color GRAY`	128, 128, 128
`public final static Color LIGHT_GRAY`	192, 192, 192
`public final static Color DARK_GRAY`	64, 64, 64

`Color` constants and their RGB values.

Color Control

Method	Description
`Color` *constructors and methods*	
`public Color( int r, int g, int b )`	Creates a color based on red, green and blue components expressed as integers from 0 to 255.
`public Color( float r, float g, float b )`	Creates a color based on red, green and blue components expressed as floating-point values from 0.0 to 1.0.
`public int getRed()`	Returns a value between 0 and 255 representing the red content.
`public int getGreen()`	Returns a value between 0 and 255 representing the green content.
`public int getBlue()`	Returns a value between 0 and 255 representing the blue content.

Method	Description
`Graphics` *methods for manipulating* `Colors`	
`public Color getColor()`	Returns `Color` object representing current color for the graphics context.
`public void setColor( Color c )`	Sets the current color for drawing with the graphics context.

`Color` methods and color-related `Graphics` methods.

JColorChooser

- Package javax.swing provides the JColorChooser GUI component that enables application users to select colors.
- JColorChooser static method showDialog creates a JColorChooser object, attaches it to a dialog box and displays the dialog.
    - Returns the selected Color object, or null if the user presses Cancel or closes the dialog without pressing OK.
    - Three arguments—a reference to its parent Component, a String to display in the title bar of the dialog and the initial selected Color for the dialog.
- Method setBackground changes the background color of a Component.

Drawing methods in Graphics class

- Graphics class provides methods for drawing three types of graphical objects:

1. **Text strings**: via the drawString() method. (System.out.println() prints to the system console, not to the graphics screen)
2. **Vector-graphic primitives and shapes**: via methods draw---() and fill---(), where --- could be Line, Rect, Oval, Arc, PolyLine, RoundRect, or 3DRect
3. **Bitmap images**: via the drawImage() method

Drawing methods in Graphics class

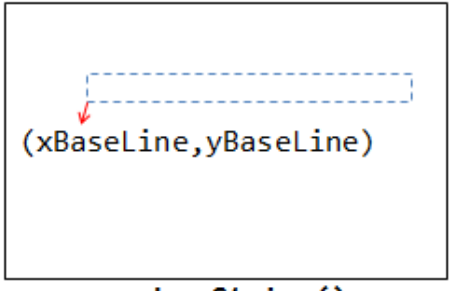
- Drawing (or printing) texts on the graphics screen:

  drawString(String str, int xBaselineLeft, int yBaselineLeft);

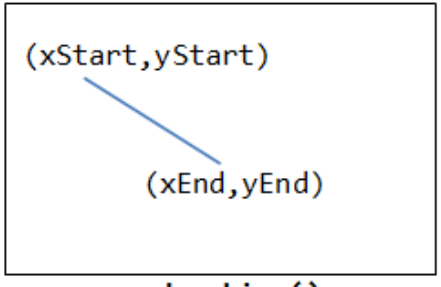
- Drawing lines:

  drawLine(int x1, int y1, int x2, int y2);

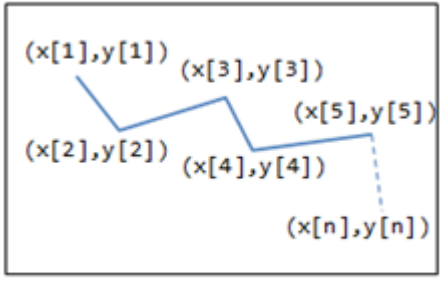
  drawPolyline(int[] xPoints, int[] yPoints, int numPoint);



drawString()

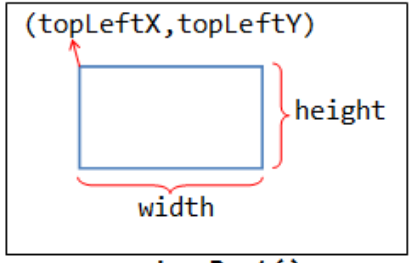


drawLine()



drawPolyline()

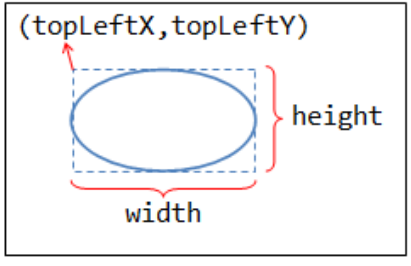
Drawing methods in Graphics class



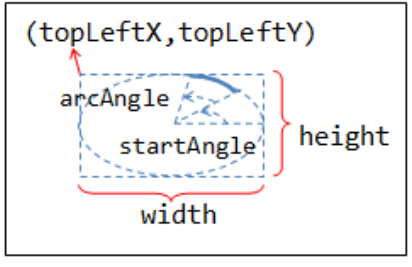
drawRect()

Drawing primitive shapes:

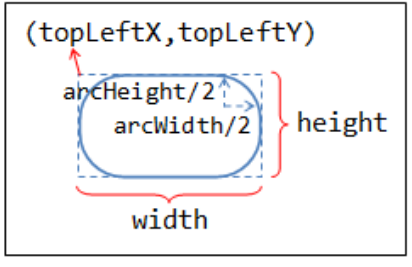
- drawRect(int xTopLeft, int yTopLeft, int width, int height);
- drawOval(int xTopLeft, int yTopLeft, int width, int height);
- drawArc(int xTopLeft, int yTopLeft, int width, int height, int startAngle, int arcAngle);
- draw3DRect(int xTopLeft, int, yTopLeft, int width, int height, boolean raised);
-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



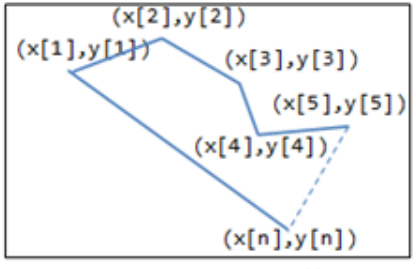
drawOval()



drawArc()



drawRoundRect()



drawPolygon()

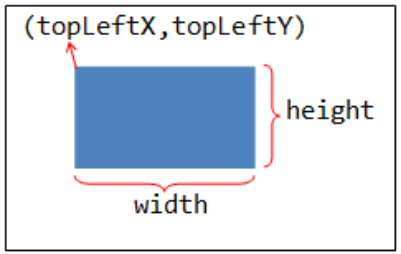
Drawing methods in Graphics class

Filling primitive shapes:

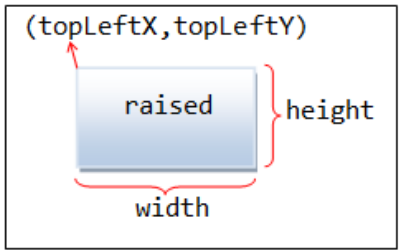
- fillRect(int xTopLeft, int yTopLeft, int width, int height);
- fillOval(int xTopLeft, int yTopLeft, int width, int height);
- fillArc(int xTopLeft, int yTopLeft, int width, int height, int startAngle, int arcAngle);
- fill3DRect(int xTopLeft, int, yTopLeft, int width, int height, boolean raised);
- 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

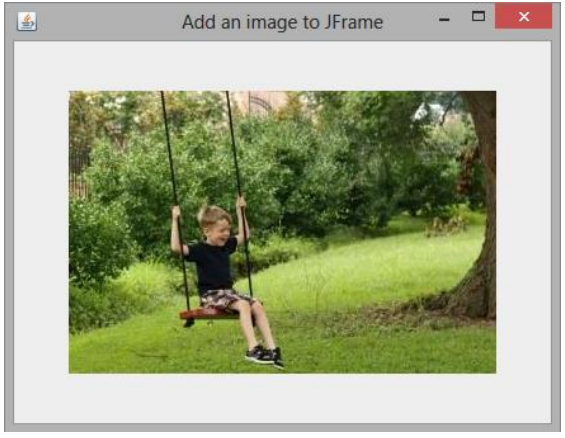


fillRect()



fill3DRect()

Displaying images

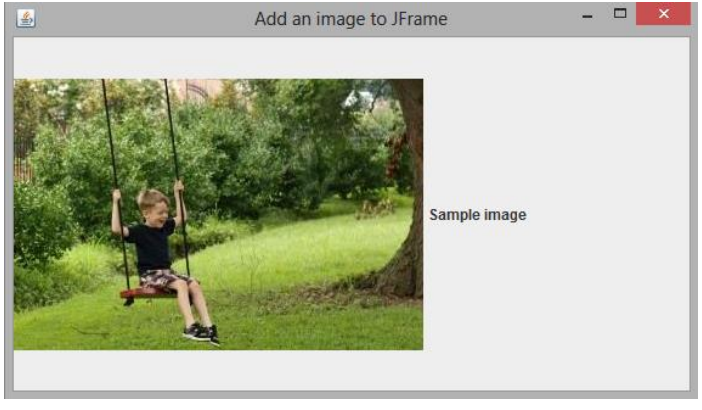


- Displaying images in JLabel

```
ImageIcon icon = new ImageIcon(URL);
JLabel label=new JLabel(icon);
```

- Displaying an image and text both in JLabel

```
ImageIcon icon = new ImageIcon(URL);
JLabel label=new JLabel("Sample image ",icon, SwingConstants.LEFT);
```

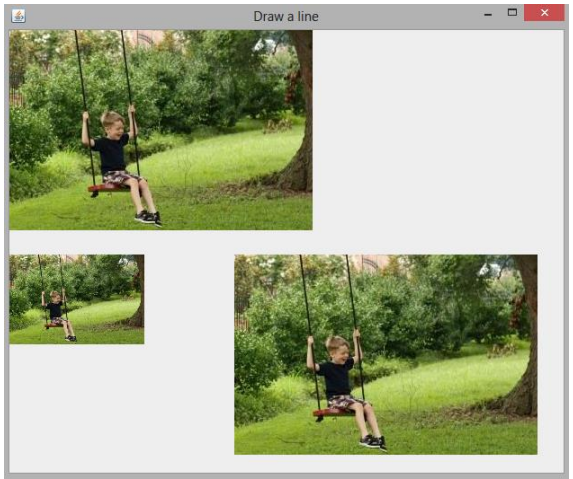


Displaying images

Drawing (or Displaying) images in Graphics:

- 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

```
public void paint(Graphics g){
        ImageIcon icon = new ImageIcon("C:/Users/roshan/Desktop/Capture.jpg");
        Image img = icon.getImage();
        g.drawImage(img, 0,0, this);
        g.drawImage(img, 250,250, this);
        g.drawImage(img, 0, 250,150, 100,this);
}
```



KeyEvent and KeyListener

- Key events in java are generated when a key is pressed, the key is typed and a key is released.
- The key Listener interface is handling key events and is defined in java.awt.event package.
- Each of the key event handling methods takes a keyEvent object is its arguments.
- A key event object contains information about the key event.

KeyEvent and KeyListener

```
public class MyClass  implements KeyListener {
    public void keyTyped(KeyEvent e) {
        // Invoked when a key has been typed.
    }
    public void keyPressed(KeyEvent e) {
        // Invoked when a key has been pressed.
    }
    public void keyReleased(KeyEvent e) {
        // Invoked when a key has been released.
    }
}
```

MouseEvent and MouseListener

- The mouse event handling in Java has two methods for interfaces.
- There are "MouseListeners" and "MouseMotionListeners".
- Both Listener interfaces handling mouse events are defined in java.awt.event package.
- These methods are called when the mouse interacts with components.
- Each of the mouse event-handling methods takes a mouse event object as its arguments.
- A mouse event object contains information about the mouse event that occurred, including the x- and y-coordinated of the location where the event occurred.

MouseEvent and MouseListener

```
public class MyClass  implements MouseListener {
   public void mouseClicked(MouseEvent e) {
      // Invoked when a mouse button is clicked (pressed and released)
   }
   public void mousePressed(MouseEvent e) {
      // Invoked when a mouse button is pressed
   }
 public void mouseReleased(MouseEvent e) {
      // Invoked when a mouse button is pressed
}
public void mouseEntered(MouseEvent e) {
      // Invoked when the mouse cursor enters a component
}
   public void mouseExited(MouseEvent e) {
      // Invoked when the mouse cursor exits a component
}
}
```

WindowEvent and WindowListener

- The Java WindowListener is notified whenever you change the state of window.
- It is notified against WindowEvent which is defined in java.awt.event package.

WindowEvent and WindowListener

```
public class MyClass  implements WindowListener {

    public void windowActivated (WindowEvent arg0) {   // Invoked
    when window is set to be active    }

    public void windowClosed (WindowEvent arg0) {    // Invoked when window is closed }

    public void windowClosing (WindowEvent arg0) {   // Invoked
    when we attempt to close window from system menu }

    public void windowDeactivated (WindowEvent arg0) {    // Invoked when window is not active

    public void windowDeiconified (WindowEvent arg0) {    //
    Invoked   when window is modified from minimized to normal state }

    public void windowIconified(WindowEvent arg0) {     // Invoked
    when window is modified from normal to minimized state  }

    public void windowOpened(WindowEvent arg0) {    // Invoked  when window is first opened }

    }
```

Adapter Classes

- Many event-listener interfaces, such as MouseListener and MouseMotionListener, contain multiple methods.
- It's not always desirable to declare every method in an event-listener interface.
- For instance, an application may need only the mouseClicked handler from MouseListener or the mouseDragged handler from MouseMotionListener.
- Interface WindowListener specifies seven window event-handling methods.
- For many of the listener interfaces that have multiple methods, packages java.awt.event and javax.swing.event provide event-listener adapter classes.

Adapter Classes

- An **adapter class** implements an interface and provides a default implementation (with an empty method body) of each method in the interface.
- An adapter class can extended to inherit the default implementation of every method and subsequently override only the method(s) that are needed for event handling.
- So, If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it saves code.
- The adapter classes are found in java.awt.event, java.awt.dnd and javax.swing.event packages.

Adapter Classes

Event-adapter class in `java.awt.event`	Implements interface
`ComponentAdapter`	`ComponentListener`
`ContainerAdapter`	`ContainerListener`
`FocusAdapter`	`FocusListener`
`KeyAdapter`	`KeyListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`WindowAdapter`	`WindowListener`

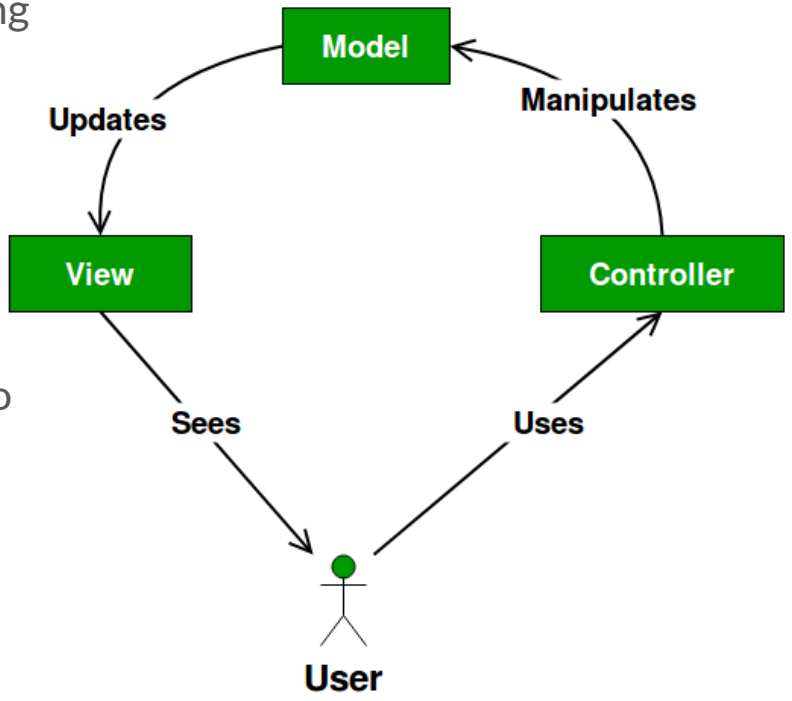
Event-adapter classes and the interfaces they implement in package `java.awt.event`.

MVC

- One of the best ways of working with a GUI is to use the "Model-View-Controller" code structure.
- MVC structure is split into three separate sections.
- **M**odel - Contains the data for your program, along with the logic, methods and functions to manipulate this data.
- **V**iew - Means of displaying the data within the model. Probably a GUI, but could be audio, printouts or any kind of thing.
- **C**ontroller - Maps the users actions in the view to model updates.

MVC

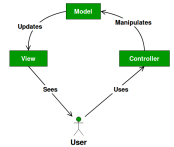
- Model - Contains the data for your program, along with the logic, methods and functions to manipulate this data.
- View - Means of displaying the data within the model. Probably a GUI, but could be audio, printouts or any kind of thing.
- Controller - Maps the users actions in the view to model updates.



MVC

```
class Student {
   private String rollNo;
   private String name;
    public String getRollNo(){return rollNo;}
    public String getName(){ return name;}
    public void setRollNo(String rollNo) {
      this.rollNo = rollNo;
   }
    public void setName(String name){
      this.name = name;
   }
}
```

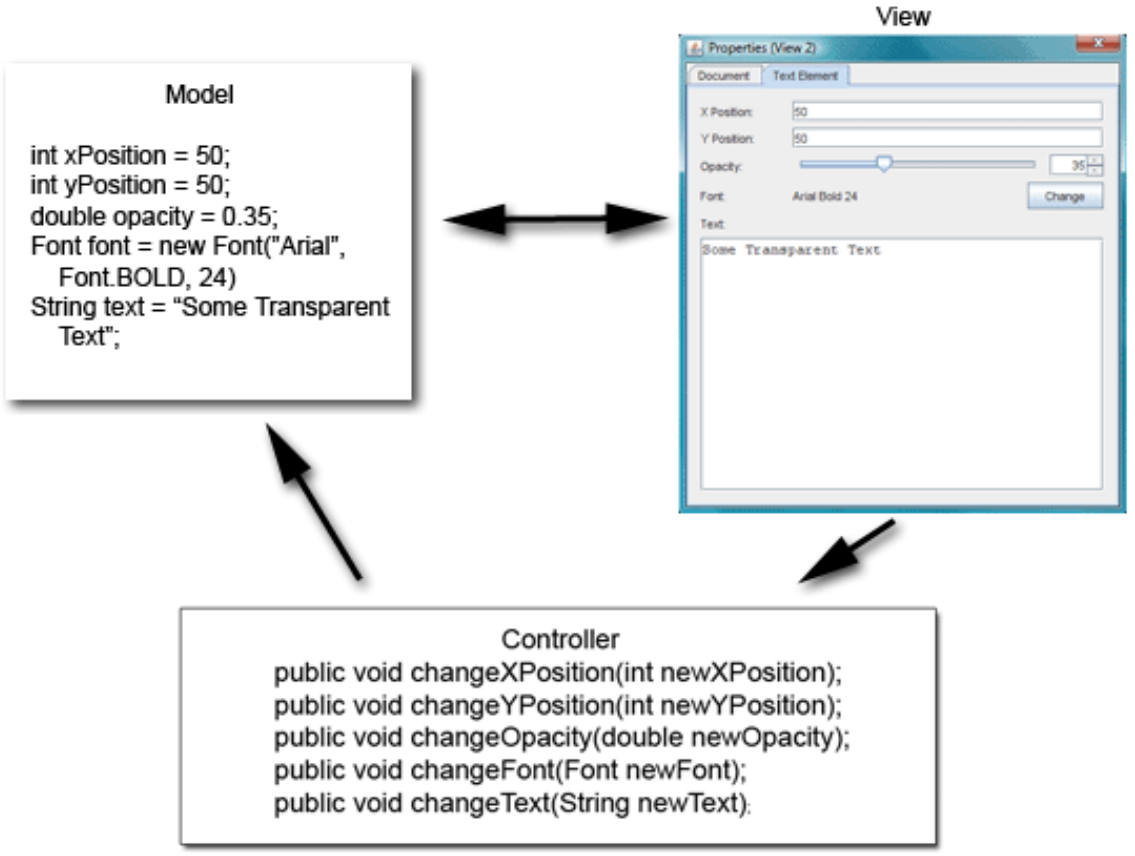
```
class StudentView {
   public void printStudentDetails(String studentName, String
studentRollNo){
      System.out.println("Student: ");
      System.out.println("Name: " + studentName);
      System.out.println("Roll No: " + studentRollNo);
   }
}
```



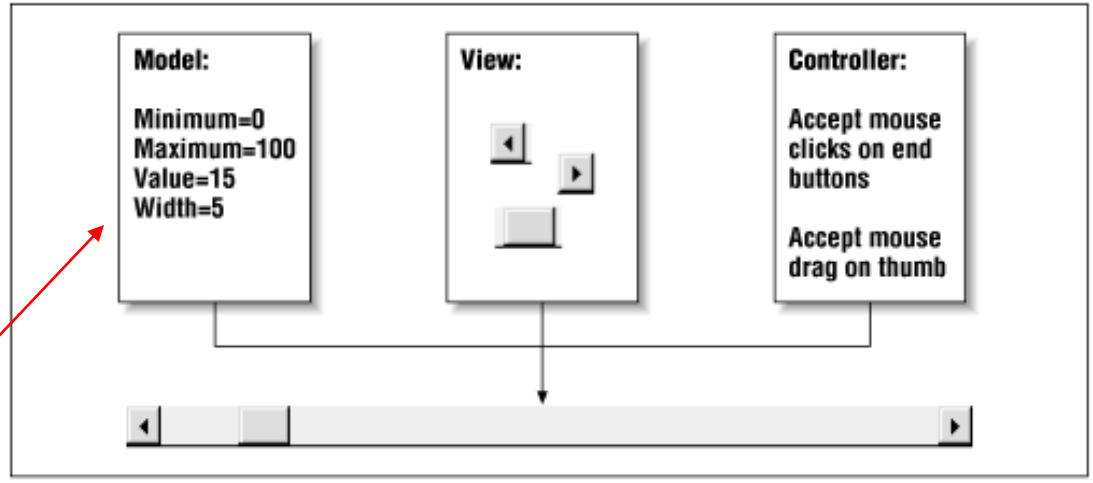
MVC

```
class StudentController {
   private Student model;
   private StudentView view;
   public StudentController(Student model, StudentView view) {
      this.model = model;
      this.view = view;
   }
  public void setStudentName(String name){
      model.setName(name);
   }
   public String getStudentName() {
      return model.getName();
   }
  public void setStudentRollNo(String rollNo) {
      model.setRollNo(rollNo);
   }
    public String getStudentRollNo(){
      return model.getRollNo();
   }
public void updateView() {
      view.printStudentDetails(model.getName(),
model.getRollNo());
   }
}
class MVCPattern {
   public static void main(String[] args){
      Student model  =
retriveStudentFromDatabase();
       StudentView view = new StudentView();
       StudentController controller = new
StudentController(model, view);
       controller.updateView();
       controller.setStudentName("Vikram Sharma"
       controller.updateView();
   }
    private static Student
retriveStudentFromDatabase(){
      Student student = new Student();
      student.setName("Lokesh Sharma");
      student.setRollNo("15UCS157");
      return student;
   }
}
```

MVC

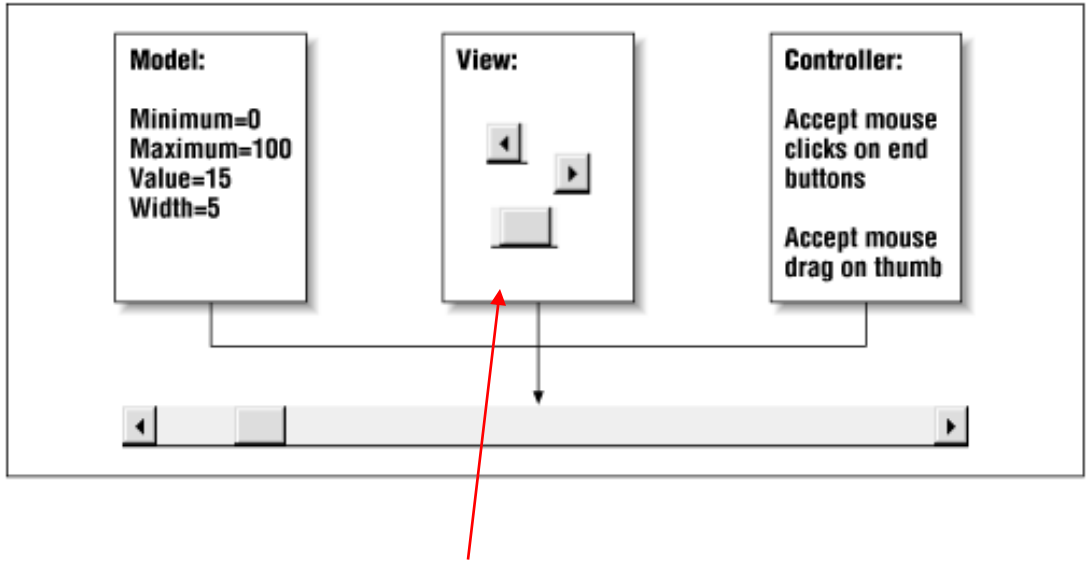

View
Model
int xPosition = 50;
int yPosition = 50;
double opacity = 0.35;
Font font = new Font("Arial",
Font.BOLD, 24)
String text = "Some Transparent
Text";
Properties (View 2)
Document
Text Element
X Position:
50
Y Position:
50
Opacity:
35
Font:
Arial Bold 24
Change
Text:
Some Transparent Text
Controller
public void changeXPosition(int newXPosition);
public void changeYPosition(int newYPosition);
public void changeOpacity(double newOpacity);
public void changeFont(Font newFont);
public void changeText(String newText);

MVC



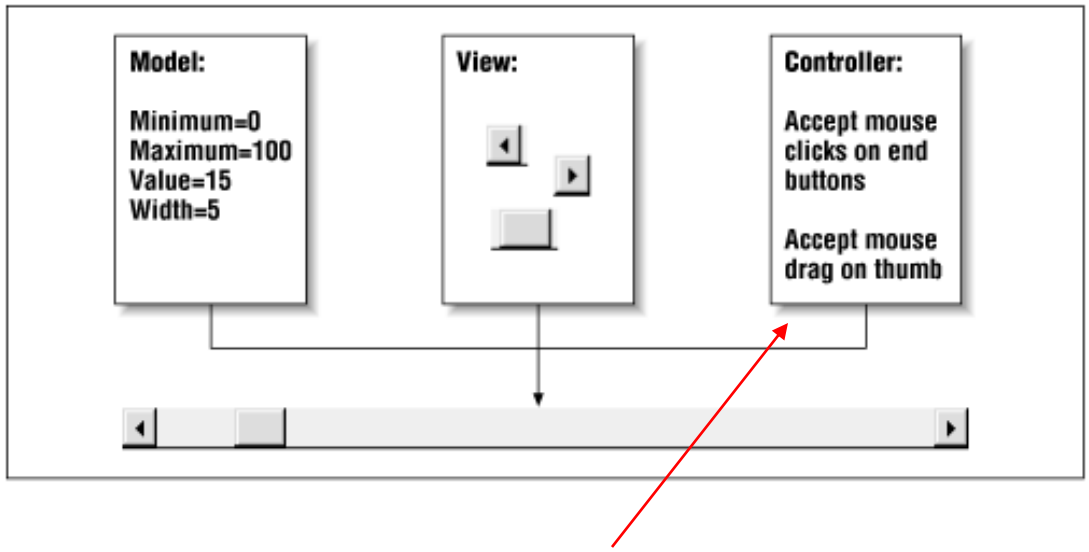
- The scrollbar uses the information in the model to determine how far into the scrollbar to render the thumb and how wide the thumb should be.
- Note that the model specifies this information relative to the minimum and the maximum. It does not give the position or width of the thumb in screen pixels—the view calculates that.

MVC



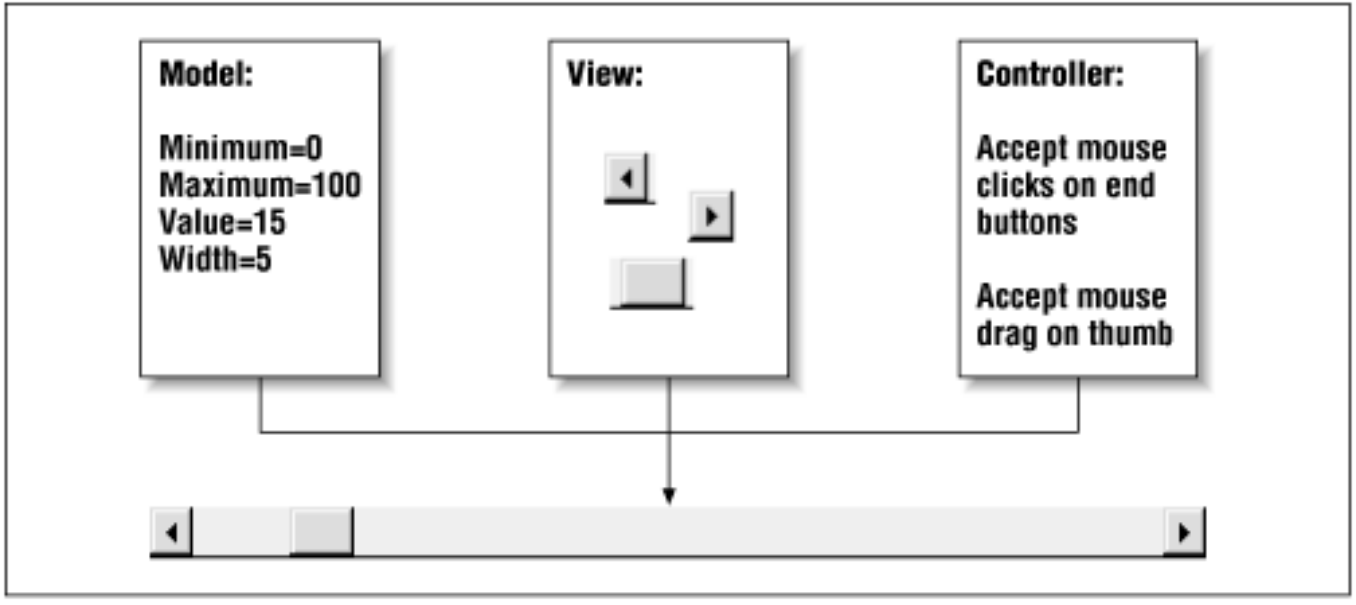
- The view determines exactly where and how to draw the scrollbar, given the proportions offered by the model.
- The view knows whether it is a horizontal or vertical scrollbar, and it knows exactly how to shadow the end buttons and the thumb.

MVC



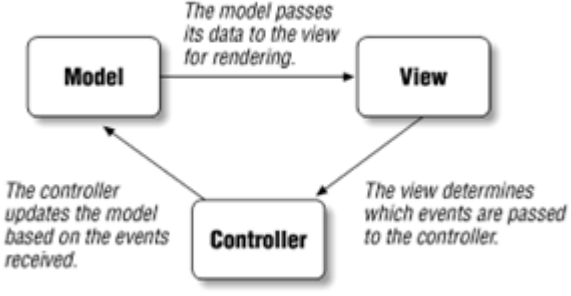
- Finally, the controller is responsible for handling mouse events on the component.
- The controller knows, for example, that dragging the thumb is a legitimate action for a scroll bar, and pushing on the end buttons is acceptable as well.

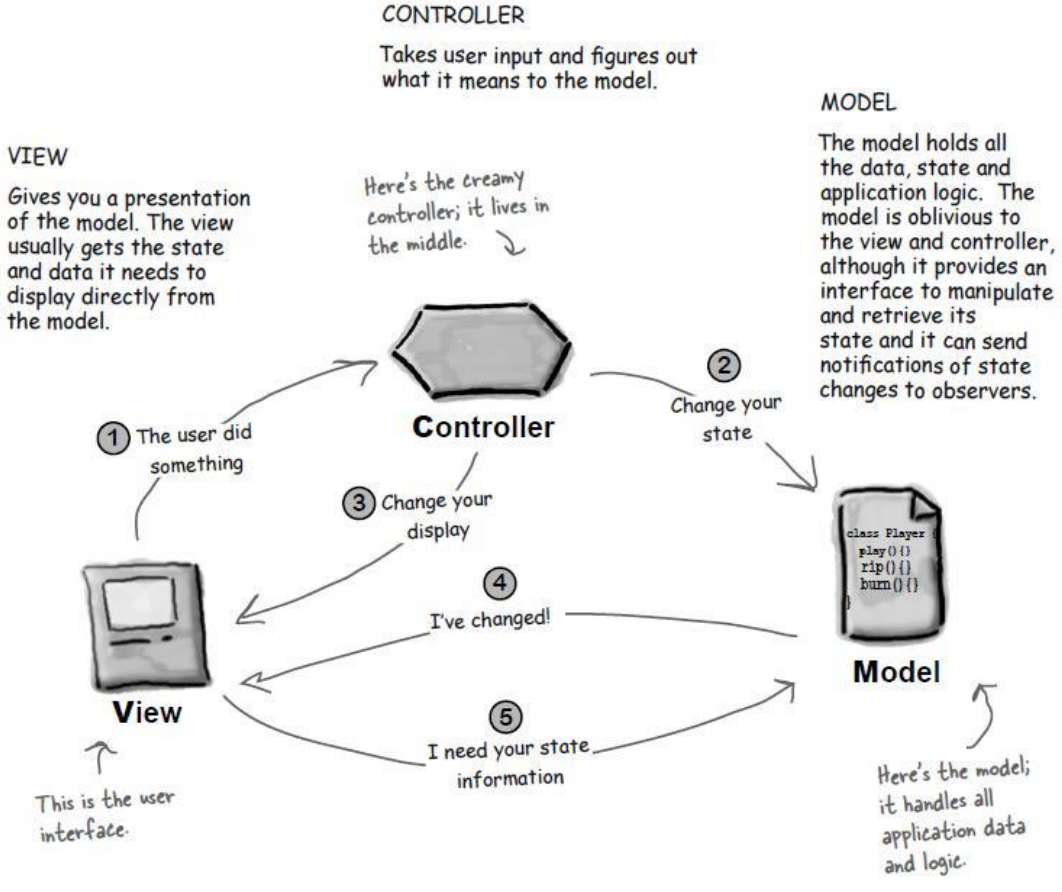
MVC



The result is a fully functional MVC scrollbar.

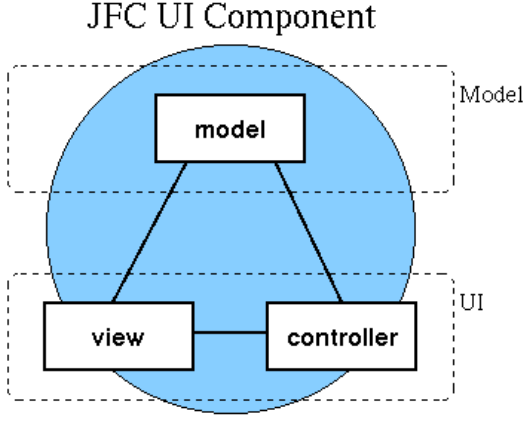
MVC Interaction





MVC in Swing

- Swing actually makes use of a simplified variant of the MVC design called the model-delegate .
- This design combines the view and the controller object into a single element that draws the component to the screen and handles GUI events known as the UI delegate.
- Bundling graphics capabilities and event handling is somewhat easy in Java, since much of the event handling is taken care of in AWT.



MVC

Pros

- Multiple developers can work simultaneously on the model, controller and views.
- MVC enables logical grouping of related actions on a controller together. The views for a specific model are also grouped together.
- Models can have multiple views.

Cons

- The framework navigation can be complex because it introduces new layers of abstraction and requires users to adapt to the decomposition criteria of MVC.
- Developers using MVC need to be skilled in multiple technologies.