Introducing Swing

BUILDING GUI WITH A POWERFUL TOOL SWING

AWT

- The AWT defines a basic set that supports limited graphical interface as it translates various components into their *peers* (platform specific).
- Look and feel is decided by platform not by java.
- Since, AWT components use native code resources, so referred as *heavyweight*.
- Use of native peers led to several problems:
  - Component might look or act differently on different platforms.
  - Look and feel of each component is fixed as it is defined by the platform.

Swing

- Swing does not replace the AWT, instead swing is built upon foundation of the AWT.
- Swing uses same event handling mechanism as the AWT.

Key features of Swing

- Platform-independent
  - This means that components are written in java and are not platform-specific.
  - Look and feel of each component is determined by Swing not by underlying Operating System(platform independent).
- Swing supports a pluggable look and feel(PLAF)
  - It is possible to change the way that a component is rendered without affecting any other aspect.
    - So it is possible to “plug in” a new look.
  - Separating out the look and feel provides significant advantages.

Components and Containers

- A GUI consists of different graphic Component objects which are combined into a hierarchy using Container objects.
- Component class:
  - an abstract class for GUI components such as menus, buttons or slider.
- A container holds a group of components.
  - Since, container is itself a component . So, a container can also hold other containers.
  - Swing allows to define a *containment hierarchy*, at the top of which is a top-level container.

Components

- Swing components are derived from JComponent class.
- JComponent inherits the AWT classes Container and Component.
- Thus, a Swing component is built on and compatible with an AWT component.
- All of Swing's component are defined in following package:
  - Javax.swing

Swing Components table

JApplet	JButton	JCheckBox	JCheckBoxMenuItem	JColorChooser
JComboBox	JComponent	JDesktopPane	JDialog	JEditorPane
JFileChooser	JFormattedTextField	JFrame	JInternalFrame	JLabel
JLayer	JLayeredPane	JList	JMenu	JMenubar
JMenuItem	JOptionPane	JPanel	JPasswordField	JPopupMenu
JProgressBar	JRadioButton	JRadioButtonMenuItem	JRootPane	JScrollBar
JScrollPane	JSeperator	JSlider	JSpinner	JSplitPane
JTabbedPane	JTable	JTextArea	JTextField	JTextPane
JTogglebutton	JToolBar	JToolTip	JTree	JViewPort
JWindow				

Weighing Components

- Two types:
  - Lightweight :
    - Lightweight components are not dependent on native peers to render themselves. They are coded in Java.
    - JPanel
    - Do Inherit JComponent.
  - Heavyweight:
    - Heavyweight components are rendered by the host operating system. They are resources managed by the underlying window manager.
    - JFrame, JApplet, JWindow, JDialog
    - Heavyweight components in Swing are also called as top level container.

Three Types of GUI Classes

1. Containers

   JFrame, JPanel, JApplet

2. Components

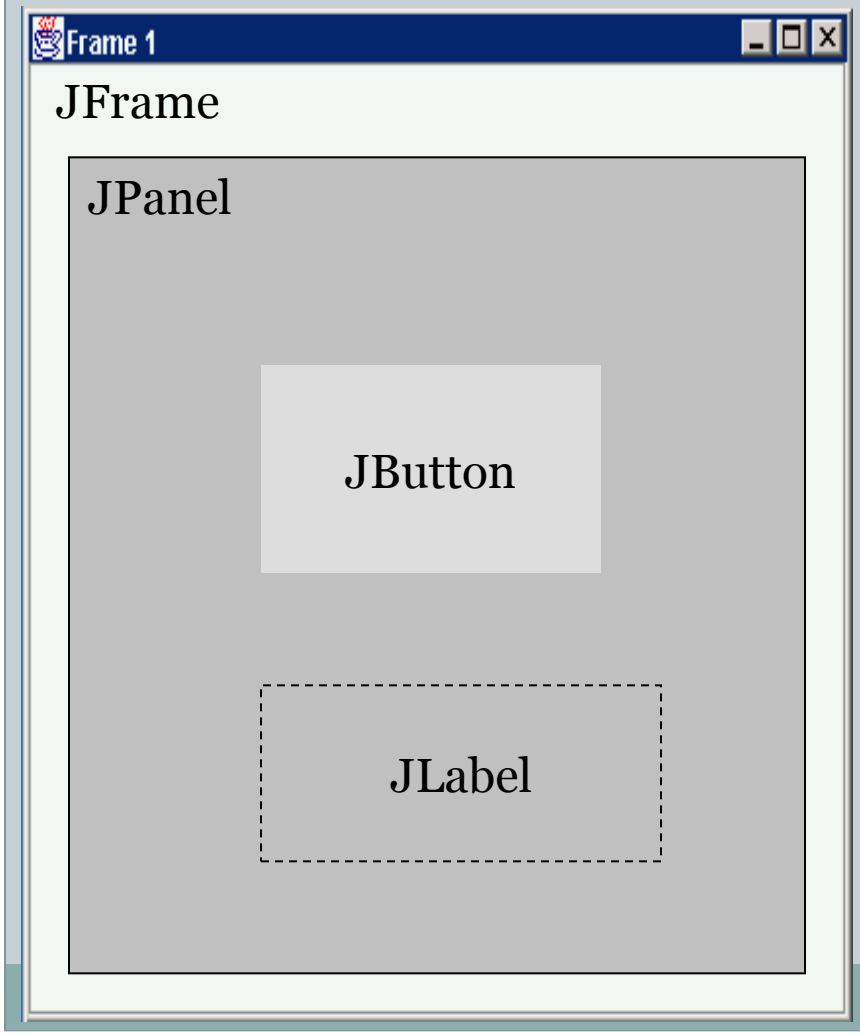
   JButton, JTextField, JComboBox, JList, etc.

3. Helpers

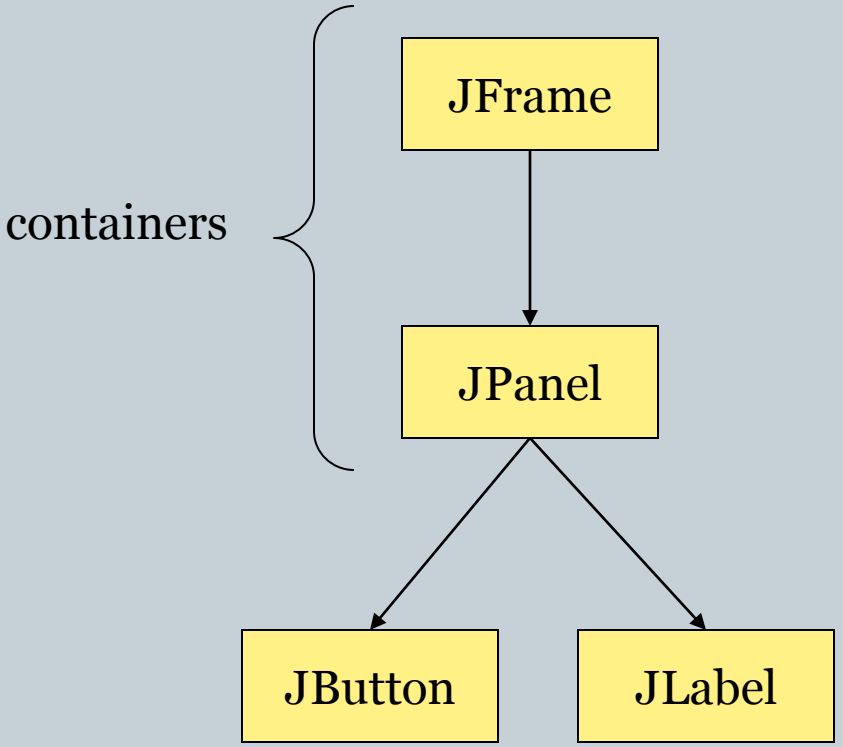
   Graphics, Color, Font, Dimension, etc.

Anatomy of an Application GUI

GUI



Internal structure



A Simple Swing Example

This package contains the components and models defined by Swing.

Will create a container and a rectangular window with complete title bar

```
import javax.swing.*;
class SwingDemo
{
        SwingDemo()
        {       //Create a new JFrame container
                JFrame jfrm = new JFrame("A Simple GUI Application");

                //Give the frame an initial size.
                jfrm.setSize(300,400);

                //Terminate the application when user closes the application
                jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                //Create a text-based label
                JLabel jlab = new JLabel("Swing means a Powerful GUIs.");

                //Add the label to the content pane
                jfrm.add(jlab);
                //Display the frame.
                jfrm.setVisible(true);
        }
```

setSize() method sets the dimensions of the window in terms of pixels.

A simplest and easy to use component.

This method allows to display the window using '*true*' arg.

A Simple Swing Example

```
public static void main(String arg[])
{
        //Create the frame on the event dispatching thread
        SwingUtilities.invokeLater(new Runnable()
        {
                public void run()
                {
                        new SwingDemo();
                }
        });
}
}
```

Ques: *why on event dispatching thread not on main thread?*
In general, Swing programs are event driven. Like when a user interacts with a component, an event is generated. And that event is passed to the defined event handler. And that handler executes on event dispatching thread not on main thread. So, although event handler functions are defined by user but they are called on a thread not by the program.

A Simple Swing Example

- To enable the GUI code to be created on the event dispatching thread, SwingUtilities class is used.
- It defines two functions:
  - Static void invokeLater(Runnable obj)
    - Returns immediately
  - Static void invokeAndWait(Runnable obj)
    - Waits until obj.run() returns.

Default Close Operation

- General form of this function is:
  - void setDefaultCloseOperation(int *what*)
- The value passed *what* determines what happens when a window is closed.
- There can be several options:
  - EXIT_ON_CLOSE
  - DISPOSE_ON_CLOSE
  - HIDE_ON_CLOSE
  - DO_NOTHING_ON_CLOSE
- Sinece, these constants are implemented by JFrame.
  - jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

Event handling

- Swing components do respond to the user input and event generated by those interactions need to be handled.
- Event can also get generate without user interaction:
  - Ex:- when a timer goes off.
- Event handling mechanism used by swing is same as AWT.
- Swing uses the same events as of AWT and these events are packaged in java.awt.event.

Event Handling Example

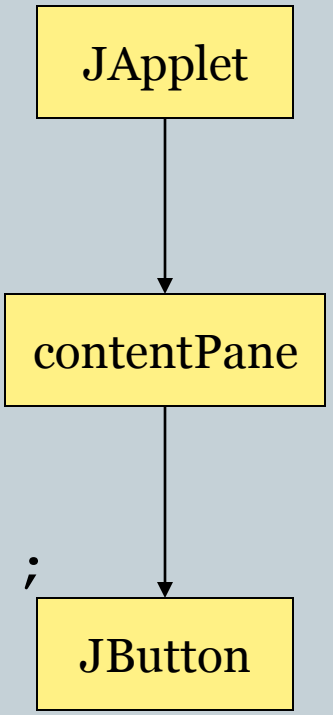
```
//create a label which is to be displayed when button is pressed
JLabel jlab = new JLabel();
//make a button
JButton jbtnAlpha = new JButton("Alpha");

//Add action listener for alpha button
jbtnAplha.addActionListener(new ActionListener())
{
        public void actionPerformed(ActionEvent ae)
        {
                jlab.setText("Alpha button is pressed");
        }
}
```

When a button is pressed, it generates an ActionEvent. So, addActionListener() method is used to add an action listener. ActionListener interface defines a method : actionPerformed().

it is an event handler that is called when a button press event has occurred.

Swing Applets

- JApplet is like a JFrame
- Already has a panel
  - Access panel with JApplet.getContentPane( )




```
import javax.swing.*;

class hello extends JApplet {
  public void init(){
      JButton b = new JButton("press me");
      getContentPane().add(b);
  }
}
```

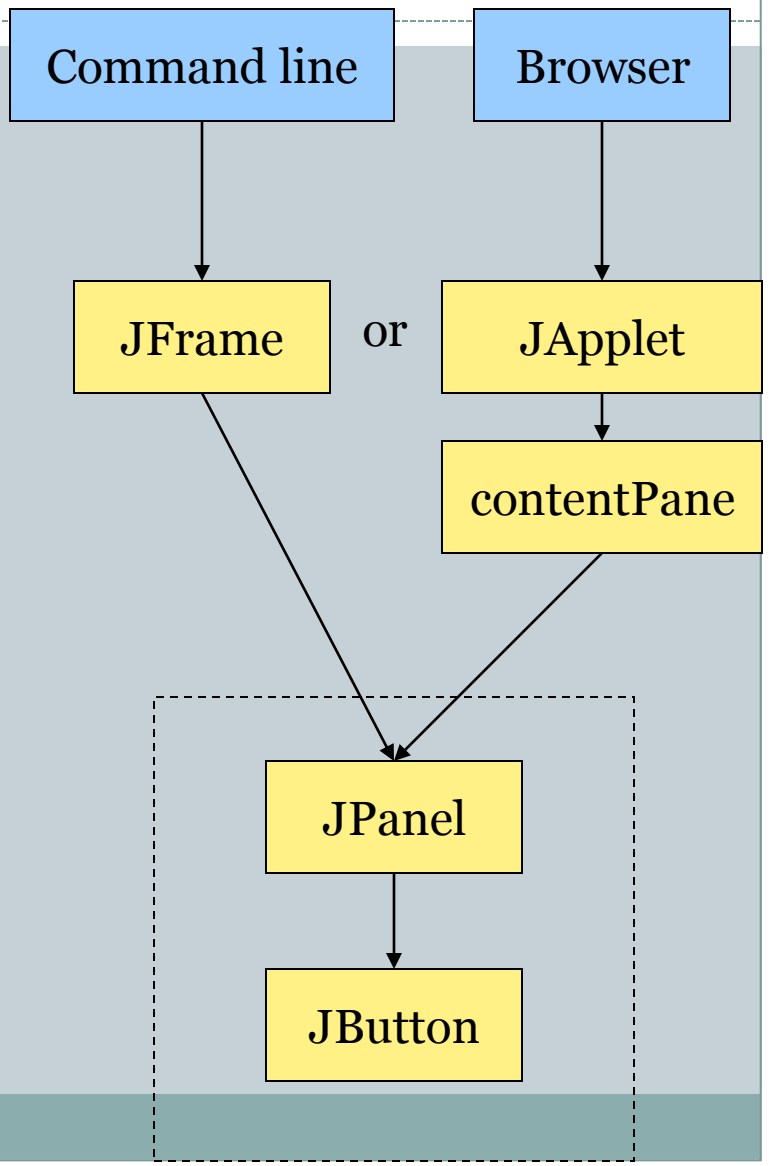
Applet Methods

- Swing Applets have the same lifecycle methods:

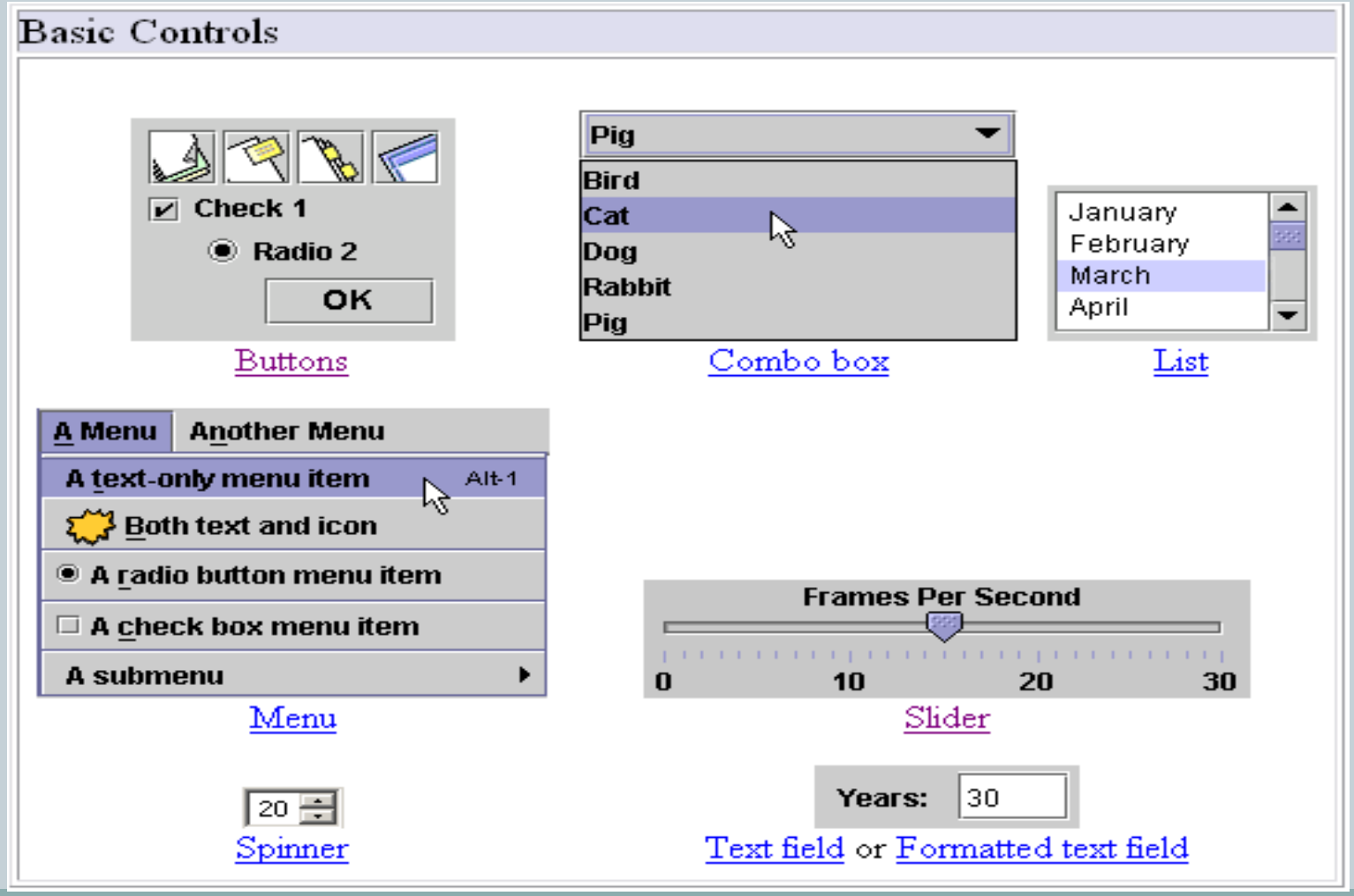
- init( ) - initialization
- start( ) - resume processing (e.g. animations)
- stop( ) - pause
- destroy( ) - cleanup
- paint( ) - redraw stuff (‘expose’ event)

Application + Applet

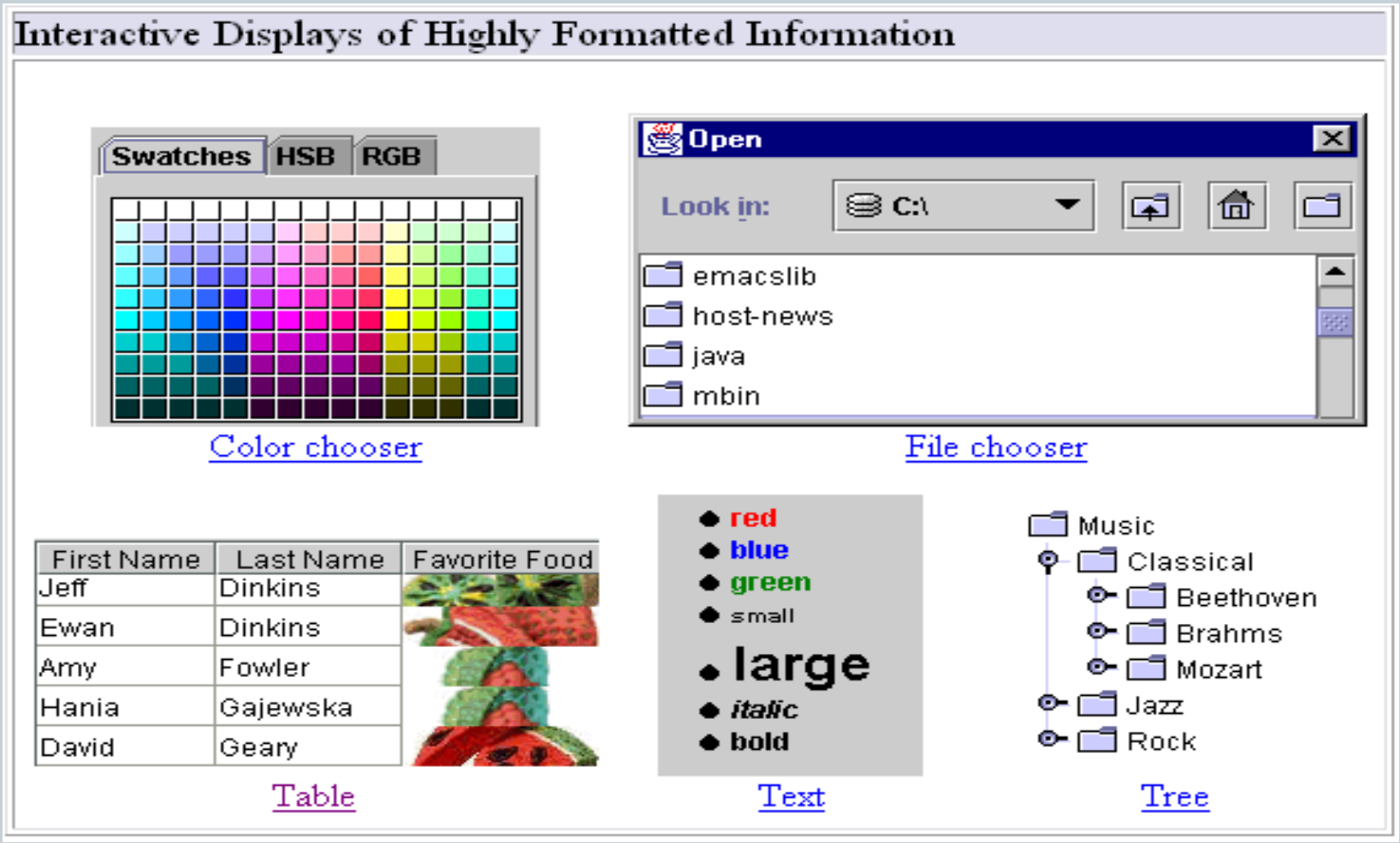
```
import javax.swing.*;

class helloApp {
   public static void main(String[] args){
          // create Frame and put my mainPanel in it
          JFrame f = new JFrame("title");
          mainPanel p = new mainPanel();
          f.add(p);
          f.setVisible(true);
   }
}

// my main GUI is in here:
class mainPanel extends JPanel {
   mainPanel(){
          setLayout(new FlowLayout());
          JButton b = new JButton("press me");
          add(b);
   }
}

class helloApplet extends JApplet {
   public void init(){
          // put my mainPanel in the Applet
          mainPanel p = new mainPanel();
          getContentPane().add(p);
   }
}
```



Swing Components



Swing Components



JFrames

- A **JFrame** is a Window with all of the adornments added.
  - **JFrame** inherits from **Frame, Window, Container, Component,** and **Object**
- A **JFrame** provides the basic building block for screen-oriented applications.

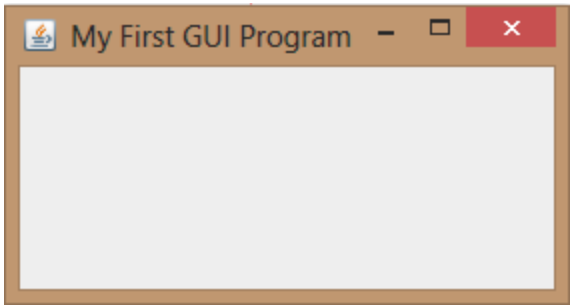
```
JFrame win = new JFrame( "title" );
```

JFrame

- Sizing a Frame
  - You can specify the size
    - Height and width given in pixels
    - The size of a pixel will vary based on the resolution of the device on which the frame is rendered.
  - The method **pack()** will set the size of the frame automatically, based on the size of the components contained in the content pane.
- Unlike a Frame, a JFrame has some notion of how to respond when the user attempts to close the window.

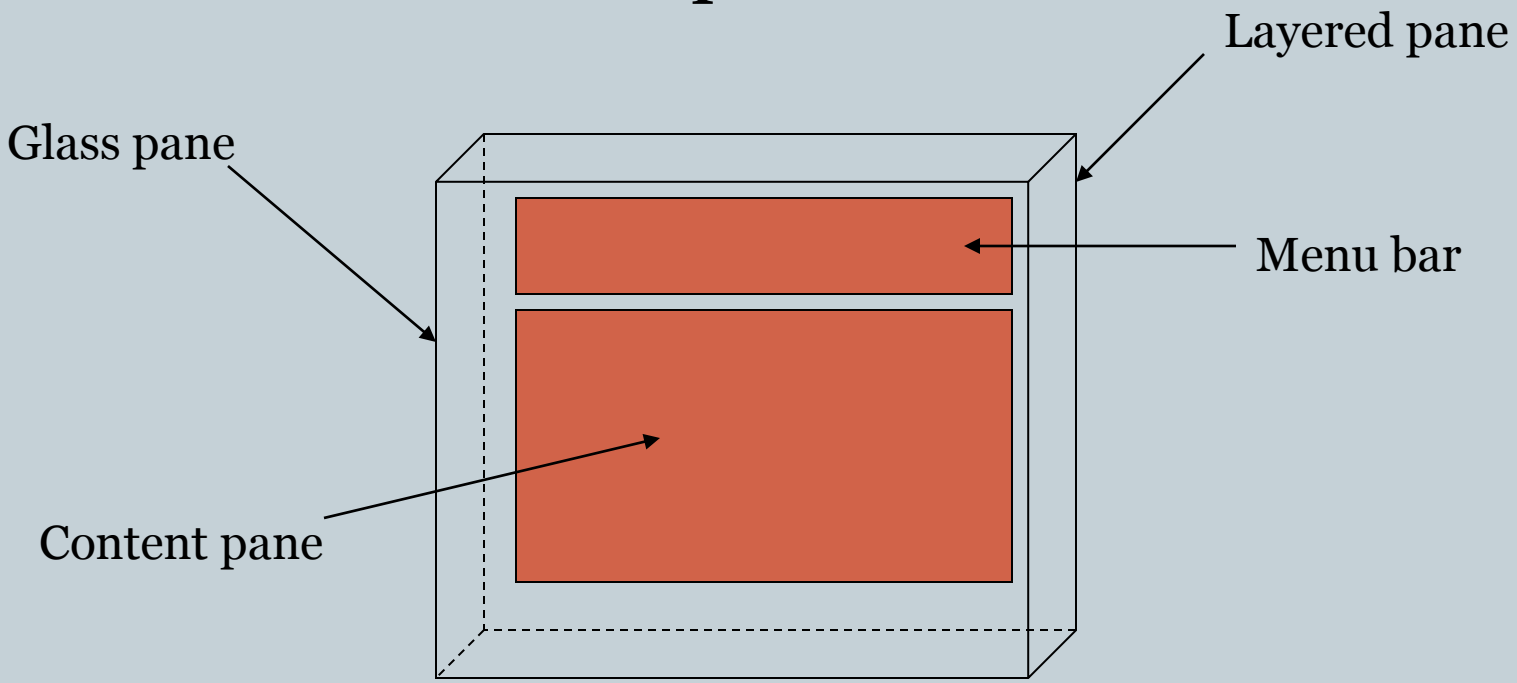
Creating a JFrame

```
    import javax.swing.*;
import java.awt.*;

public class swingframe {

    public static void main( String args[] ) {
        JFrame win = new JFrame( "My First GUI Program" );
        win.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

        win.setSize( 250,150 );
        win.setVisible(true);
    }
}
```



JFrame

- **JFrame**s have several panes:



- Components are placed in the Content Pane

Top Level Container Panes

- JRootPane is a lightweight container whose purpose is to manage the other panes.
  - Panes that comprise root pane are *glass pane, content pane, layered pane*.
- *Glass Pane* is the top-level pane, which covers all other panes and enables to manage mouse events that effect the entire container.
- *Layered Pane* is an instance of JLayeredPane that allows components to be given a depth value.
  - *This* value determines which component overlays other.
- *Content Pane* is the one with which application will interact.
  - So, when we add a button, it is added to content pane only. It is an instance of JPanel.

JLabel and ImageIcon

- Swing's easiest-to-use component.
- Is used to display text and/or an icon.
- It is a passive component as it does not respond to user input.
- Constructors defined in JLabel:
  - JLabel(Icon *icon*)
  - JLabel(String *str*)
  - JLabel(String *str*, Icon *icon*, int *align*)
- Where, *str* and *icon* are text and icon used for label and *align* argument specifies the horizontal alignment of the text and/or icon within the dimension of the label.

JLabel and ImageIcon

- ImageIcon class implements Icon and encapsulates an image.
- Thus, an object of type ImageIcon can be passed as an argument to the Icon parameter of JLabel's Constructor.
- ImageIcon's Constructor:
  - ImageIcon(String *filename*)
    - which obtains the image specified by file named *filename*.

JLabel and ImageIcon

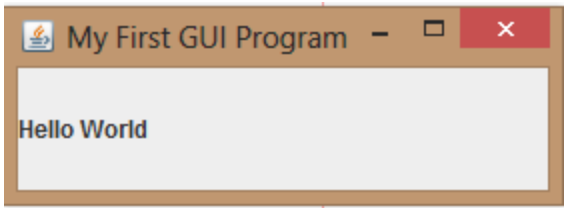
- The icon and text associated with the label can be obtained by following methods:
  - Icon getIcon()
  - String getText()
- The icon and Text associated with a label can be set by following methods:
  - void setIcon(Icon *icon*)
  - void setText(String *str*)
    - setText() method can be used to change the text of label.
- ImageIcon class implements the following methods:
  - int getIconHeight()
    - Returns the height of the icon in pixels.
  - int getIconWidth()
    - Returns the width of the icon in pixels.

Example

- Create a frame
  - JFrame jfrm = new JFrame();
- Create an icon
  - ImageIcon ii = new ImageIcon(“india.jpg”);
- Create a label
  - JLabel jlab= new JLabel(“India”,ii,JLabel.CENTER);
- Add this label to the content pane.
  - jfrm.add(jlab);

Hello World

```
import javax.swing.*;

public class swinglabel {
    public static void main( String args[] ) {
        JFrame win = new JFrame( "My First GUI Program" );
        win.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        win.setSize(100,100);
        JLabel label = new JLabel( "Hello World" );
        win.add( label );
        win.setVisible(true);
    }
}
```



JTextField

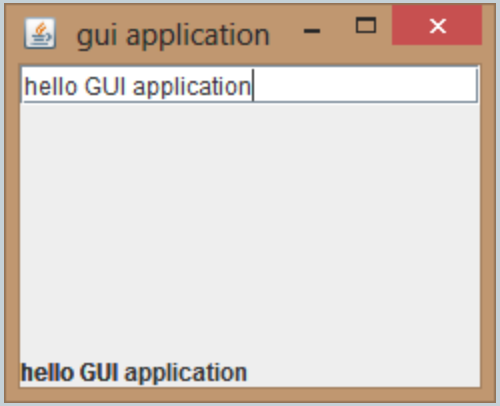
- JTextField is the simplest Swing text component.
- Used to edit one line of text.
- JTextField's constructors:
  - JTextField(int *cols*)
  - JTextField(String *str*, int *cols*)
  - JTextField(String *str*)
    - Where, *str* is the string to be initially presented, *col* represents number of columns in text field.
- If no string is specified, text field is initially empty.

JTextField

- Several events can generate in response to user interaction with text field:
  - **ActionEvent:** fired when user presses ENTER.
  - **CaretEvent:** fired when position of cursor changes.
- To obtain the text currently in the text field, call getText() function.

Example

```
public class swingtext implements ActionListener
{
     JFrame jfrm;
     JTextField jtf;
     public static void main(String args[])
    {
        swingtext st=new swingtext();
        st.Makegui();


     }
     void Makegui()
     {
          jfrm = new JFrame("gui application");
          jtf=new JTextField();
          jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          jfrm.setSize(200,200);
          jfrm.add(jtf,BorderLayout.NORTH);
          jtf.addActionListener(this);
          jfrm.setVisible(true);
     }
      public void actionPerformed(ActionEvent ae)
             {
                  JLabel jlab= new JLabel(jtf.getText());
                  jfrm.add(jlab,BorderLayout.SOUTH);
                    jfrm.setVisible(true);
             }

}
```



The Swing Buttons

- Swing defines four types of buttons:
  - JButton
  - JToggleBuuton
  - JCheckBox
  - JRadioButton
- Methods to control behaviour of buttons:
  - void setDisabledIcon(Icon *di*)
  - void setPressedIcon(Icon *pi*)
  - void setSelectedIcon(Icon *si*)
  - void setRolloverIcon(Icon *ri*)
    - *di,pi,si,ri* icon would be displayed when button is disabled, pressed,selected, mouse is positioned over a button.
- Text associated with button can be read and written using:
  - String getText()
  - Void setText(String str)

JButtons

- JButton extends Component , displays a string, and delivers an **ActionEvent** for each mouse click.
- In addition to text, JButtons can also display icons.
- Constructors:
  - JButton(Icon *icon*)
  - JButton(String *str*)
  - JButton(String str,Icon *icon*)
- Using ActionEvent object passed to actionPerformed() method , *action command* associated with button can be obtained.

JButtons

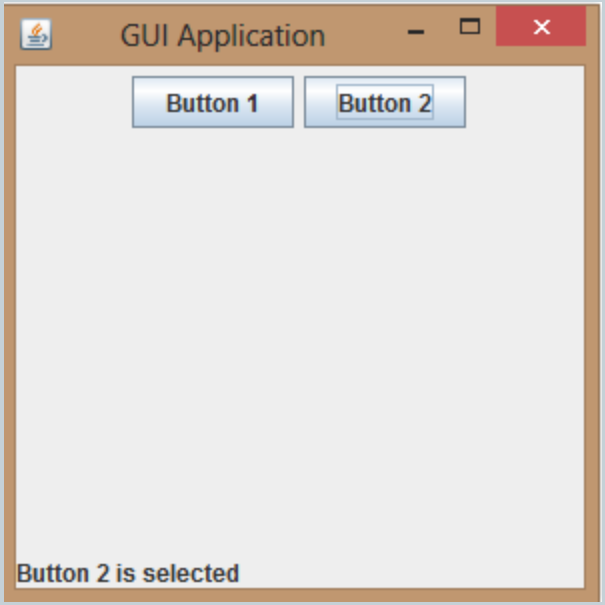
- String getActionCommand()
  - The action command identifies the pressed button.
- void setActionCommand(String *str*)
  - Used to set the action command
- When using two or more buttons within same app, the action command helps to determine which button has been pressed.

New about JButton

- Unlike Button, Jbutton provides an ability to add any of the component inside a button itself using
  - Add(Component *comp*) method

Example

```
public class SwingDemo implements ActionListener
{
  JFrame jfrm;
  JLabel jlab;
    void makegui()
    {
        jfrm=new JFrame("GUI Application");
        jfrm.setSize(300,300);
        JPanel jp1=new JPanel();
        JButton button = new JButton("Button 1");
        JButton button1 = new JButton("Button 2");
        //jp1.setLayout(new FlowLayout());
        button.setSize(5,5);
        button1.setSize(5,5);
        jp1.add(button);
        jp1.add(button1);
        jfrm.add(jp1,BorderLayout.NORTH);
        button.addActionListener(this);
        button1.addActionListener(this);
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jlab= new JLabel("no button is selcted");
        jfrm.add(jlab,BorderLayout.SOUTH);
        jfrm.setVisible(true);
    }
    public void actionPerformed(ActionEvent ae)
     {
                jlab.setText(ae.getActionCommand()+" is selected");
     }
```

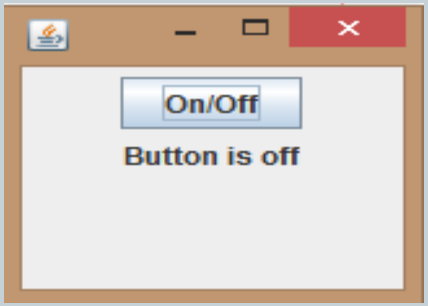


JToggleButton

- Toggle button have 2 states:
  - Pushed
    - When button is pressed, it says pressed
  - Released
    - When button is pressed second time, it releases(pops up).
- Each time a toggle button is pushed, it toggles b/w two states.
- JToggleButton(String *str*)
  - Creates a toggle button that contains the text pressed in str.
- JToggleButton also generates an item event.
  - getItem() method can be called to obtain a reference to generated event.
  - isSelected() method is used to determine a toggle button's state that generated event.

Example

```
class swingtogglebutton implements ItemListener
{
    JFrame jfrm;
   JLabel jlab;
   JToggleButton jtbn;
   void makegui()
   {
       jfrm=new JFrame();
       jfrm.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
       jfrm.setSize(100,150);
       jlab=new JLabel("Button is off");
       jtbn=new JToggleButton("On/Off");
       jtbn.addItemListener(this);
       jfrm.add(jtbn);
       jfrm.add(jlab);
       jfrm.setVisible(true);
   }
   public void itemStateChanged(ItemEvent ie)
   {
       if(jtbn.isSelected())
           jlab.setText("Button is on");
       else
           jlab.setText("Button is off");
   }
public static void main(String args[])
{
 swingtogglebutton stbn=new swingtogglebutton();
    stbn.makegui();
}
```



Check Boxes

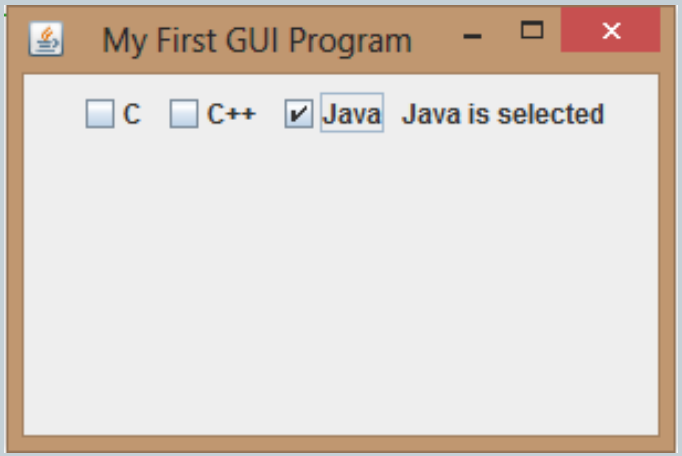
- JCheckBox class provides the functionality of a check box.
- Some of the constructors for checkbox are:
  - JCheckBox(String *str*)
    - Creates a check box that has text specified by str as a label.
  - JCheckBox(Icon *i)*
  - JCheckBox(Icon *i, boolean state)*
  - JCheckBox(String *str, boolean state)*
  - JCheckBox(String *str, Icon i)*
  - JCheckBox(String *str, Icon i, boolean state)*
- When a user selects or deselects a check box, an ItemEvent is generated.

Example

```
public class SwingCheckBox implements ItemListener
{
    JFrame win;
    JLabel jlab;
    public static void main( String args[] ) {

        SwingCheckBox st=new SwingCheckBox();
        st.MakeGui();
    }
    void MakeGui()
    {
        win = new JFrame( "My First GUI Program" );
        win.setLayout(new FlowLayout());
        win.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JCheckBox cb=new JCheckBox("C");
        win.add(cb);
        cb.addItemListener(this);
        cb=new JCheckBox("C++");
        win.add(cb);
        cb.addItemListener(this);
        cb=new JCheckBox("Java");
        win.add(cb);
        jlab = new JLabel("button is off");
        win.add(jlab);
        cb.addItemListener(this);
        win.setVisible(true);
        win.setSize(200,200);
    }
```

Example

```
public void itemStateChanged(ItemEvent ie)
    {
        JCheckBox cb= (JCheckBox)ie.getItem();
        if(cb.isSelected())
            jlab.setText(cb.getText()+" is selected");
        else
            jlab.setText(cb.getText()+" is cleared");
    }
```



Radio Buttons

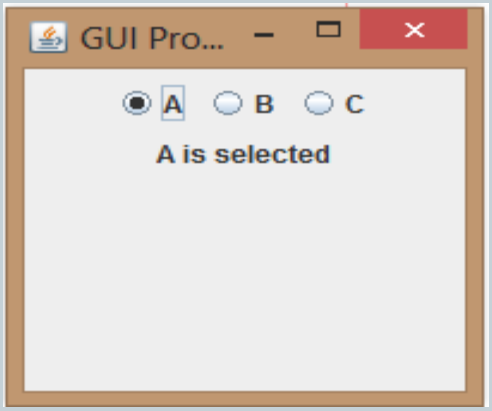
- A group of mutually exclusive buttons, in which only one can be selected at any point of time.
- Some of the constructor for Radio Buttons are:
  - JRadioButton(String *str*)
  - JRadioButton(Icon *i)*
  - JRadioButton(Icon *i, boolean state)*
  - JRadioButton(String *str, boolean state)*
  - JRadioButton(String *str, Icon i)*
  - JRadioButton(String *str, Icon i, boolean state)*
    - Where, *str* is the label for the button, icon *i* specifies an image, *state* specifies default state for the button.

Radio Buttons

- Radio buttons must be configured into a group which is created by ButtonGroup class.
- JRadioButton generates action events, item events each time the button selection changes.
- ActionListener interface defines a function actionPerformed() in which getActionCommand() can be called to get button selected.

Example

```
void makegui()
{
    jfrm = new JFrame( "GUI Program" );
    jfrm.setLayout(new FlowLayout());
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    ButtonGroup bg= new ButtonGroup();
    JRadioButton b1 = new JRadioButton("A");
    jfrm.add(b1);
    bg.add(b1);
    b1.addActionListener(this);
    b1 = new JRadioButton("B");
    jfrm.add(b1);
    bg.add(b1);
    b1.addActionListener(this);
    b1 = new JRadioButton("C");
    jfrm.add(b1);
    bg.add(b1);
    b1.addActionListener(this);
    jlab= new JLabel("No radio button is selected.");
    jfrm.add(jlab,BorderLayout.SOUTH);
    jfrm.setVisible(true);
    jfrm.setSize(200,200);
}
public void actionPerformed(ActionEvent ae)
{
     jlab.setText(ae.getActionCommand()+" is selected");
}
```



JTabbed Pane

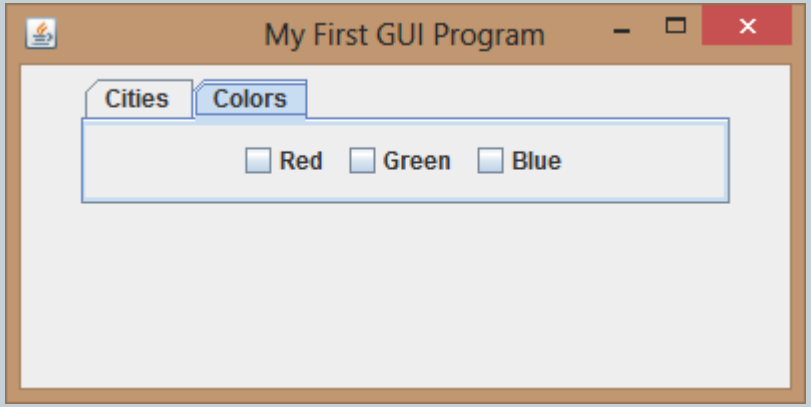
- It manages a set of components by linking them with tabs.
- Selecting a tab causes the components associated with that tab to come to the forefront.
- The general procedure to use a tabbed pane is:
  1) Create an instance of JTabbedPane.
  2) Add each tab by calling addTab().
     1) void addTab(String *name*, Component *comp*)
     2) Where, *name* is name of the tab, and *comp* is the component that should be added to the tab.
  3) Add the tabbed pane to the content pane.

Example

```
public class SwingTabbedPane
{
      JLabel jlab;
      JFrame jfrm;
      JTabbedPane jtp;
    public static void main( String args[] ) {
        SwingTabbedPane stb = new SwingTabbedPane();
        stb.makegui();

        }
    void makegui()
            {
                jfrm = new JFrame( "My First GUI Program" );
        jfrm.setLayout(new FlowLayout());
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jtp=new JTabbedPane();
        jtp.addTab("Cities",new CitiesPanel());
        jtp.addTab("Colors",new ColorPanel());
        jfrm.add(jtp);
        jlab=new JLabel("nothing is selected");
        jfrm.setVisible(true);
        jfrm.setSize(400,200);
       }
```

Example

```
class CitiesPanel extends JPanel
{   CitiesPanel()
    {
        JButton b1=new JButton("New York");
        add(b1);
        JButton b2=new JButton("London");
        add(b2);
        JButton b3=new JButton("Delhi");
        add(b3);
        JButton b4=new JButton("Tokyo");
        add(b4);
    }
}
class ColorPanel extends JPanel
{   ColorPanel()
    {
        JCheckBox cb1 = new JCheckBox("Red");
        add(cb1);
        JCheckBox cb2 = new JCheckBox("Green");
        add(cb2);
        JCheckBox cb3 = new JCheckBox("Blue");
        add(cb3);
    }   }
```



JScrollPane

- A lightweight container that automatically handles the scrolling of another component.
- Component being scrolled can be an individual component such as a table contained within JPanel.
- Viewable area of scroll pane is called the *viewport*.
- Some of its constructors are shown here:
  - JScrollPane(Component *comp)*
  - JScrollPane(int *vsb, int hsb)*
  - JScrollPane(Component *comp, int vsb, int hsb)*
- Here, *comp* is the component to be added to the scroll pane and *vsb,hsb* are int constants.

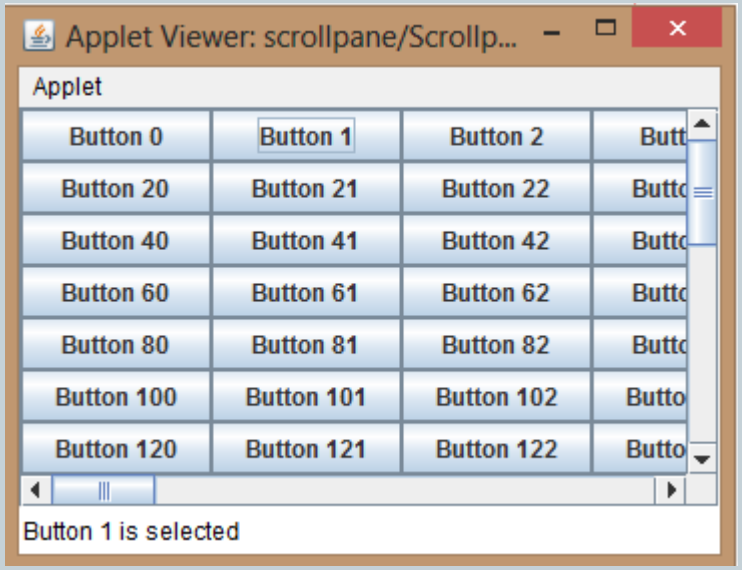
JScrollPane

- These constants are defined by the ScrollPaneConstants interface.
- Few of them are:
  - HORIZONTAL_SCROLLBAR_ALWAYS
    - Always provide horizontal scroll bar
  - HORIZONTAL_SCROLLBAR_AS_NEEDED
    - Provide horizontal scroll bar, if needed
  - VERTICAL_SCROLLBAR_ALWAYS
    - Always provide vertical scroll bar
  - VERTICAL_SCROLLBAR_AS_NEEDED
    - Provide vertical scroll bar, if needed
- Steps to be followed to use a scroll pane:
  1) Create the component to be scrolled.
  2) Create an instance of JScrollPane, passing to it the object to scroll.
  3) Add the scroll pane to the content pane.

Example

```
public class Scrollpane extends JApplet implements ActionListener
{    public void init()
    {
        // Get content pane
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        // Add 400 buttons to a panel
        JPanel jp = new JPanel();
        JButton jb;
        jp.setLayout(new GridLayout(20, 20));
        int b = 0;
        for(int i = 0; i < 20; i++) {
            for(int j = 0; j < 20; j++) {
                jb=new JButton("Button " + b);
                jp.add(jb);
                jb.addActionListener(this);
                ++b;
            }
        }
        int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
        int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
        JScrollPane jsp = new JScrollPane(jp, v, h);
        contentPane.add(jsp, BorderLayout.CENTER);
    }
    public void actionPerformed(ActionEvent ae)
    {
        this.showStatus(ae.getActionCommand()+" is selected");
    }
}
```

Example


Applet Viewer: scrollpane/Scrollp...
Applet
Button 0
Button 1
Button 2
Button 20
Button 21
Button 22
Button 40
Button 41
Button 42
Button 60
Button 61
Button 62
Button 80
Button 81
Button 82
Button 100
Button 101
Button 102
Button 120
Button 121
Button 122
Button 1 is selected

JList

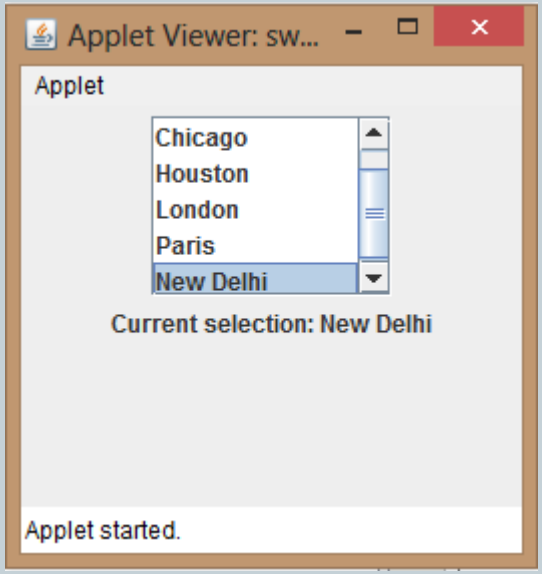
- In Swing, the basic list class is called Jlist.
- It supports selection of one or more items from a list.
- Items in a Jlist were represented as Object references.
  - class JList<E>
    - Where E represents the type of the items in the list.
- JList generates a ListSelectionEvent when the user makes or changes a selection.
  - Which is handled by ListSelectionListener.
- ListSelectionEvent and ListSelectionListener are packaged in *javax.swing.event*.
- Listener specifies only one method **valuechanged().**

JList

- JList allows the user to select multiple ranges of items using setSelectionMode().
- Void setSelectionMode(int *mode*)
  - *mode* specifies the selection mode. Which, takes one of these values defined by ListSelectionModel:
    - SINGLE_SELECTION
      - Select only a single item.
    - SINGLE_INTERVAL_SELECTION
      - *Select one range of items*
    - MULTIPLE_INTERVAL_SELECTION
      - *Selects multiple ranges of items within a list.*
- Index of first item selected can be obtained by calling getSelectedIndex().
- And value associated with this index by getSelectedValue()

Example

```
public class Swinglist  extends JApplet implements ListSelectionListener
{
    JList<String> jlist;
    JLabel jlab;
    JScrollPane jscrlp;
    String cities[] = {"New York","Chicago","Houston",
                       "London","Paris","New Delhi"};
    public void init()
    {

        setLayout(new FlowLayout());
        jlist = new JList<String>(cities);
        jlist.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        jscrlp =  new JScrollPane(jlist);
        jscrlp.setPreferredSize(new Dimension(120,90));
        jlab = new JLabel("Choose a city");
        jlist.addListSelectionListener(this);
        add(jscrlp);
        add(jlab);
    }
    public void valueChanged(ListSelectionEvent lse)
    {
        int indx=jlist.getSelectedIndex();
        if(indx!=-1)
            jlab.setText("Current selection: "+cities[indx]);
        else
            jlab.setText("Choose a city");
            }
}
```



JComboBox

- A combination of a text field and drop-down list which allows a user to select among items in list.
- Combo box normally displays only one entry.
- Combo box is declared as follows:
  - class JComboBox<E>
    - E represents the type of items.
- Items are added to the list of choices via the addItem() method, whose signature is:
  - void addItem(Object *obj)*
    - Here, *obj* is the object to be added to the combo box.

JComboBox

- Combo Box can generate:
  - ActionEvent
    - When user selects an item from the list
  - ItemEvent
    - When the state of selection changes, which occurs when an item is selected or deselected.
    - So, changing a selection generates two item events: one for selected item, another for deselected one.
- **getSelectedItem()** can be used to obtain the selected item.

JTree

- A tree is a component that presents a hierarchical view of data.
- A user can expand or collapse individual subtrees in the display.
- Trees are implemented in Swing by the JTree class, which extends JComponent.
- Some of its constructors are shown here:
  - JTree(Hashtable *ht)*
    - creates a tree in which each element of the hash table *ht* is a child node.
  - JTree(Object *obj[ ])*
    - Each element of the array *obj* is a child node
  - JTree(TreeNode *tn)*
    - tree node *tn is the* root of the tree
  - JTree(Vector *v)*
    - the last form uses the elements of vector *v as* child nodes.

JTree

- Jtree generates a variety of events, three of them are:
  - **TreeExpansionEvent**
    - Occurs when a node is expanded or collapsed.
  - **TreeSelectionEvent**
    - Generated when the user selects or deselects a node.
  - **TreeModelEvent**
    - When some data of the tree changes.
- Listeners of these events are TreeExpansionListener, TreeSelectionListener, TreeModelListener.
- Path to the selected object can be obtained by calling **getpath()** method.

Trees

- The *getPathForLocation( )* method is used to translate a mouse click on a specific point of the tree to a tree path. Its signature is shown here:
  - TreePath getPathForLocation(int x, int y)
    - Here, x and y are the coordinates at which the mouse is clicked. The return value is a TreePath object that encapsulates information about the tree node that was selected by the user.
- The *TreeNode* interface declares methods that obtain information about a tree node.
  - For example, it is possible to obtain a reference to the parent node.
- The *MutableTreeNode* interface extends *TreeNode* which declares methods that can insert and remove child nodes or change the parent node.

Trees

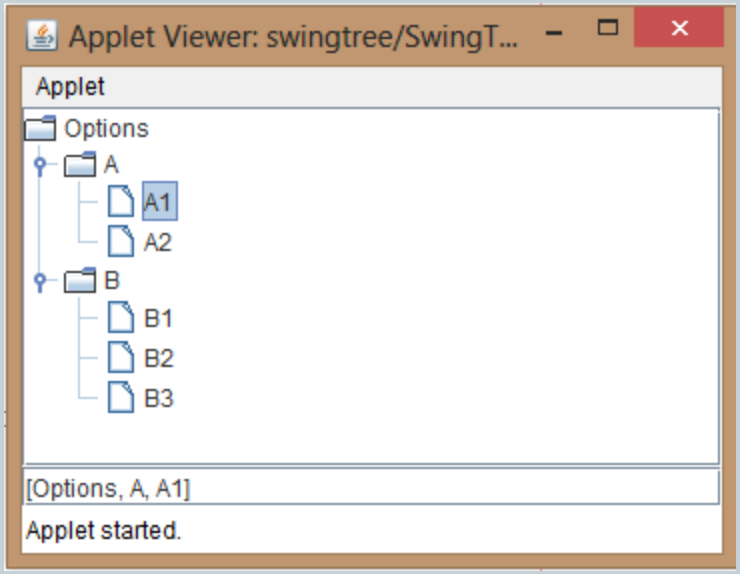
- The *DefaultMutableTreeNode* class implements the *MutableTreeNode* interface.
  - It represents a node in a tree. One of its constructors is shown here:
    - DefaultMutableTreeNode(Object *obj)*
      - Here, *obj* is the object to be enclosed in this tree node.
- To create a hierarchy of tree nodes, **add()** can be used.
- Steps to be followed to use a tree:
  1) Create an instance of JTree.
  2) Create a JScrollPane and specify the tree as the object to scroll.
  3) Add the tree to the scroll pane.
  4) Add the scroll pane to the content pane.

Example

```
public class SwingTree extends JApplet
{
        JTree tree;
         JTextField jtf;
        public void init()
         {
      // Get content pane
     Container contentPane = getContentPane();
// Set layout manager
     contentPane.setLayout(new BorderLayout());
// Create top node of tree
DefaultMutableTreeNode top = new DefaultMutableTreeNode("Options");
// Create subtree of "A"
DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
top.add(a);
DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");
a.add(a1);
DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");
a.add(a2);
// Create subtree of "B"
DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");
top.add(b);
DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("B1");
b.add(b1);
DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("B2");
b.add(b2);
DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("B3");
b.add(b3);
```

Example

```
// Create tree
tree = new JTree(top);
// Add tree to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(tree, v, h);
// Add scroll pane to the content pane
contentPane.add(jsp, BorderLayout.CENTER);

// Add text field to applet
jtf = new JTextField("", 20);
contentPane.add(jtf, BorderLayout.SOUTH);
// Anonymous inner class to handle mouse clicks
tree.addMouseListener(new MouseAdapter() {
public void mouseClicked(MouseEvent me) {
doMouseClicked(me);
}
});
    }
  void doMouseClicked(MouseEvent me)
  {
TreePath tp = tree.getPathForLocation(me.getX(), me.getY());
if(tp != null)
jtf.setText(tp.toString());
else
jtf.setText("");
}
}
```



JTable

- A *table* is a component that displays rows and columns of data.
- Cursor can be dragged on column boundaries to resize columns.
- Can also drag a column to a new position.
- Tables are implemented by the JTable class, which extends JComponent.
- One of its constructors is shown here:
  - JTable(Object *data[ ][ ], Object colHeads[ ])*
    - Here, *data is a two-dimensional array of the information to be presented, and colHeads* is a one-dimensional array with the column headings.

JTable

- JTable can generate different events, one of them is **ListSelectionEvent** .
- **ListSelectionEvent** is generated when the user selects something in the table.
- Selection can be :
  - One or more complete rows
  - One or more columns
  - One or more individual cells.
- Another type of event is **TreeModelEvent** which is fired when table's data changes in some way.

JTable

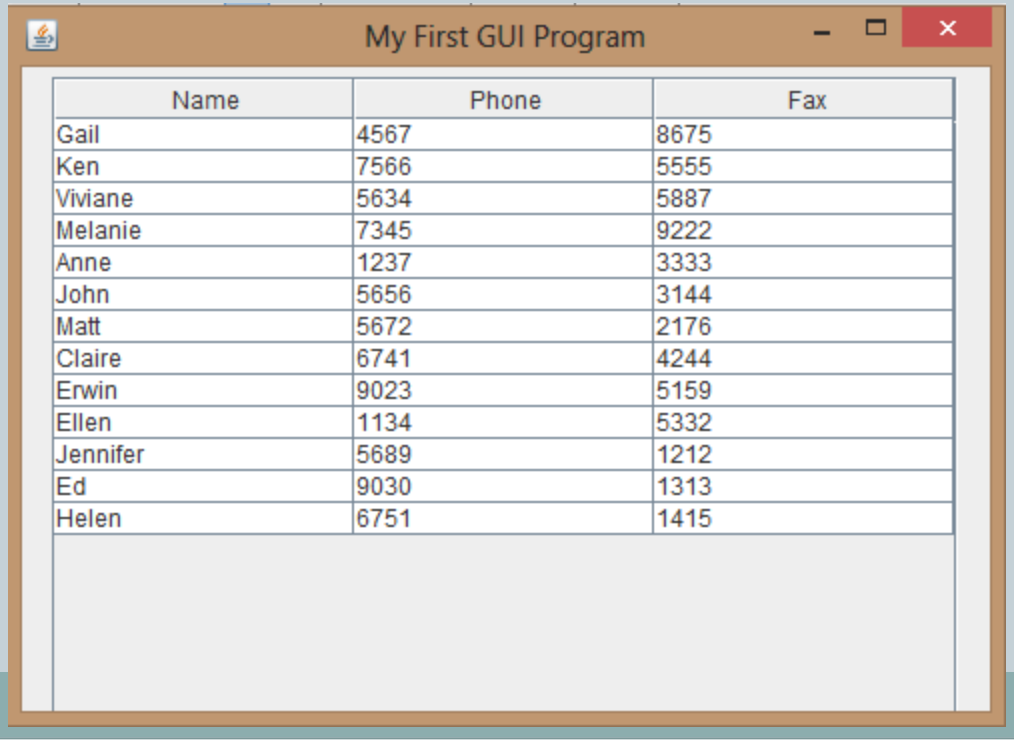
- Steps to be followed to use a table:
  1) Create an instance of JTable.
  2) Create a JScrollPane object and specify the table as the object to scroll.
  3) Add the table to the scroll pane.
  4) Add the scroll pane to the content pane

Example

```
class SwingTable extends JApplet {
    public static void main( String args[] ) {
  JFrame jfrm = new JFrame( "My First GUI Program" );
  JLabel jlab;
        jfrm.setLayout(new FlowLayout());
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        jfrm.setSize(500,500);
final String[] colHeads = { "Name", "Phone", "Fax" };
// Initialize data
final Object[][] data = {
{ "Gail", "4567", "8675" },
{ "Ken", "7566", "5555" },
{ "Viviane", "5634", "5887" },
{ "Melanie", "7345", "9222" },
{ "Anne", "1237", "3333" },
{ "John", "5656", "3144" },
{ "Matt", "5672", "2176" },
{ "Claire", "6741", "4244" },
{ "Erwin", "9023", "5159" },
{ "Ellen", "1134", "5332" },
{ "Jennifer", "5689", "1212" },
{ "Ed", "9030", "1313" },
{ "Helen", "6751", "1415" }
};
// Create the table
JTable table = new JTable(data, colHeads);
```

Example

```
// Add table to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(table, v, h);
// Add scroll pane to the content pane
jfrm.add(jsp, BorderLayout.CENTER);
jfrm.setVisible(true);
    }
}
```



Name	Phone	Fax
Gail	4567	8675
Ken	7566	5555
Viviane	5634	5887
Melanie	7345	9222
Anne	1237	3333
John	5656	3144
Matt	5672	2176
Claire	6741	4244
Erwin	9023	5159
Ellen	1134	5332
Jennifer	5689	1212
Ed	9030	1313
Helen	6751	1415

Painting in Swing

- Swing's approach to painting is built on AWT.
- The AWT class Component defines a method *paint()* that is used to draw on the surface of a component.
- Since, JComponent class in Swing inherits Component class of AWT.
- So, all Swing's lightweight components inherits the *paint()* method.
- Swing involves three distinct methods:
  - paintComponent(),paintBorder(),paintChildren()
- paintComponent() method paints the interior of the component.

Compute the Paintable Area

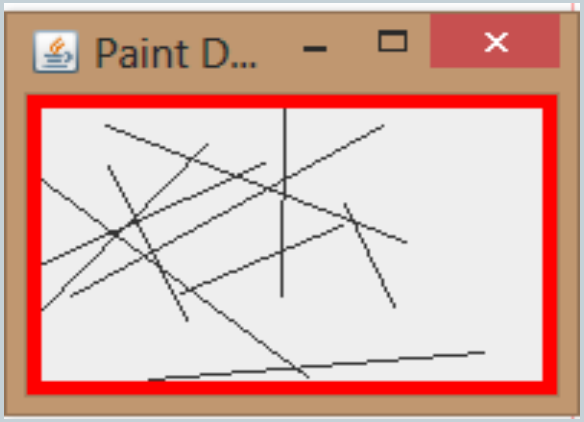
- While drawing on the surface of the component, output must restrict to the area inside border.
  - Must compute the *paintable area* of the component.
- *Paintable area = size of the component – size of the border.*
- To obtain the border width, call getInsets():
  - Insets getInsets()
    - Returns an Insets object containing dimensions of the border.
- Width and height of the component can be obtained by calling *getWidth()* and *getHeight()* functions.

Example

```
class PaintPanel extends JPanel
{
    Insets ins; //defined in awt package
    Random rand; //defined in util
    PaintPanel()
    {
        setBorder(BorderFactory.createLineBorder(Color.RED,5));
                    rand = new Random();
    }

    @Override
    protected void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        int x,y,x2,y2;
        int height=getHeight();
        int width = getWidth();
        ins=getInsets();
        for(int i=0;i<10;i++)
        {
            x=rand.nextInt(width-ins.left);
            y=rand.nextInt(height-ins.bottom);
            x2=rand.nextInt(width-ins.left);
            y2=rand.nextInt(height-ins.bottom);
            g.drawLine(x, y, x2, y2);
        }
    }
}
```

Example

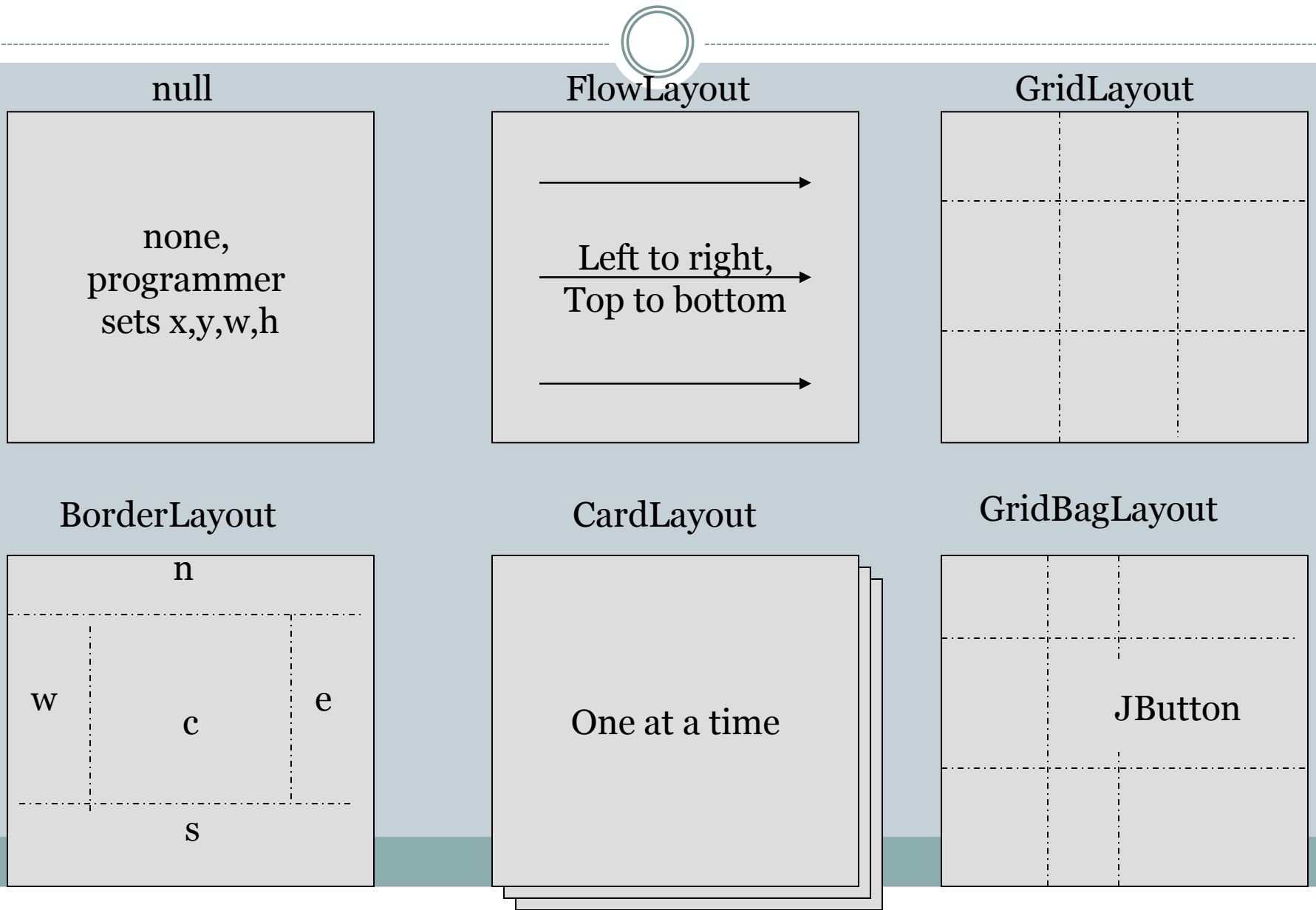
```
public class PaintDemo {

    /**
     * @param args the command line arguments
     */
    JLabel jlab;
    PaintPanel pp;
    PaintDemo()
    {
        JFrame jfrm = new JFrame("Paint Demo");
        //jfrm.pack();
        jfrm.setSize(200,150);
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pp=new PaintPanel();
        jfrm.add(pp);
        jfrm.setVisible(true);
    }
    public static void main(String[] args) {
        new PaintDemo();
    }
}
```



Layout Manager

- Layout Manager
  - An interface that defines methods for positioning and sizing objects within a container. Java defines several default implementations of LayoutManager.
- Geometrical placement in a Container is controlled by a **`LayoutManager`** object

Layout Manager Heuristics



Components, Containers, and Layout Managers

- Containers may contain components (which means containers can contain containers!!).
- All containers come equipped with a layout manager which positions and shapes (lays out) the container's components.
- Much of the action in Swing occurs between components, containers, and their layout managers.

Layout Managers

- Layouts allow you to format components on the screen in a platform-independent way
- The standard JDK provides many classes that implement the **LayoutManager** interface, including:
  - **FlowLayout**
  - **GridLayout**
  - **BorderLayout**
  - **BoxLayout**
  - **CardLayout**
  - **OverlayLayout**
  - **GridBagLayout**

Changing the Layout

1. To change the layout used in a container you first need to create the layout.
2. Then you invoke the **setLayout()** method on the container to use the new layout.

```
JPanel p = new JPanel() ;
p.setLayout( new FlowLayout() );
```

- The layout manager should be established before any components are added to the container

FlowLayout

- **FlowLayout** is the default layout for the **JPanel** class.
- When you add components to the screen, they flow left to right (centered) based on the order added and the width of the screen.
- Very similar to word wrap and full justification on a word processor.
- If the screen is resized, the components' flow will change based on the new width and height

Flow Layout

```
import javax.swing.*;
import java.awt.*;

public class ShowFlowLayout {
    public static void main( String args[] ) {
        JFrame win = new JFrame( "My First GUI Program" );
        win.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

        win.getContentPane().setLayout( new FlowLayout() );

        for ( int i = 0; i < 10; i++ ) {
            win.getContentPane().add(
                new JButton( String.valueOf( i ) ) );
        }

        win.setVisible(true);
    }
} // ShowFlowLayout
```

FlowLayout


My First GUI Program
0
1
2
3
4
5
6
7
8
9


My First GUI Program
0
1
2
3
4
5
6
7
8
9


My...
0
1
2
3
4
5
6
7
8
9

GridLayout

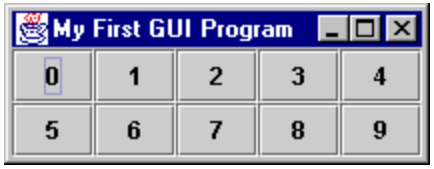
- Arranges components in rows and columns
  - If the number of rows is specified
    - columns = number of components / rows
  - If the number of columns is specified
    - Rows = number of components / columns
  - The number of columns is ignored unless the number of rows is zero.
- The order in which components are added matters
  - Component 1 → (0,0), Component 2 → (0,1), ……
- Components are resized to fit the row-column area

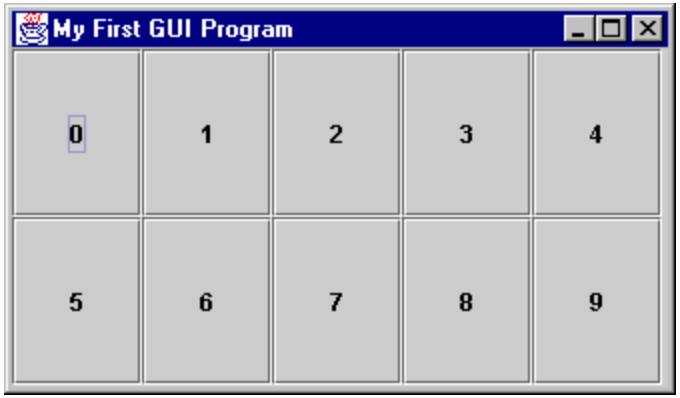
Grid Layout

```
import javax.swing.*;
import java.awt.*;

public class ShowGridLayout {
    public static void main( String args[] ) {
        JFrame win = new JFrame( "My First GUI Program" );
        win.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

        win.getContentPane().setLayout( new GridLayout( 2, 0 ) );

        for ( int i = 0; i < 10; i++ ){
            win.getContentPane().add(
                new JButton( String.valueOf( i ) ) );
        }

        win.setVisible(true);
    }
} // ShowGridLayout
```

GridLayout

```
gridLayout( 2, 4 )
```






```
gridLayout( 0, 4 )
```


```
gridLayout( 4, 4 )
```


```
gridLayout( 10, 10 )
```

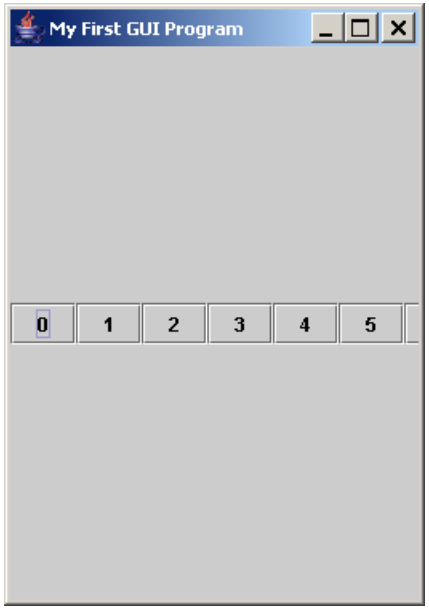
BoxLayout

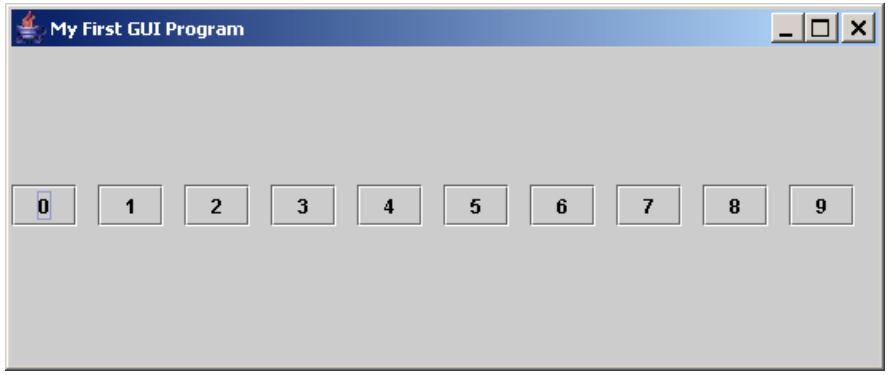
- **BoxLayout** provides an easy way to lay out components horizontally or vertically.
- Components are added in order.
- **BoxLayout** attempts to arrange components at their
  - *preferred widths* (for horizontal layout) or
  - *preferred heights* (for vertical layout).
- Static methods in **Box** class are available for "glue" and "struts."

BoxLayout example

```
import javax.swing.*;
import java.awt.*;

public class ShowBoxLayout {
    public static void main( String args[] ) {
        JFrame win = new JFrame( "My First GUI Program" );
        win.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

        win.getContentPane().setLayout(
                new BoxLayout( win.getContentPane(), BoxLayout.X_AXIS ) );

        for ( int i = 0; i < 10; i++ ){
            win.getContentPane().add( new JButton( String.valueOf( i ) ) );
            win.getContentPane().add( Box.createHorizontalGlue() );
        }

        win.pack();
        win.setVisible(true);
    }
} // ShowBoxLayout
```

BoxLayout





Note that components retain their preferred size.

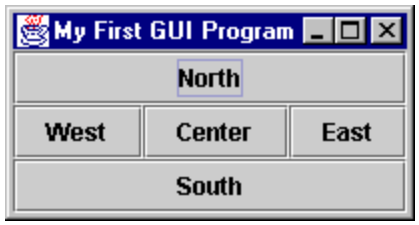
BorderLayout

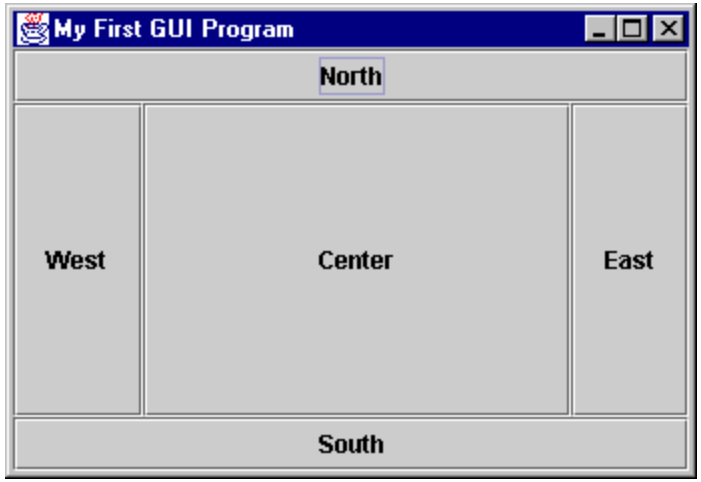
- `BorderLayout` provides 5 areas to hold components. These are named after the four different borders of the screen, North, South, East, West, and Center.
- When a Component is added to the layout, you must specify which area to place it in. The order in which components are added is not important.
- The center area will always be resized to be as large as possible

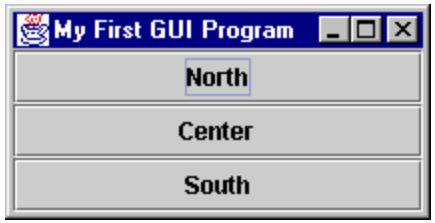
BorderLayout

```
import javax.swing.*;
import java.awt.*;

public class ShowBorderLayout {
    public static void main( String args[] ) {
        JFrame win = new JFrame( "My First GUI Program" );
        win.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

        Container content = win.getContentPane();
        content.setLayout( new BorderLayout() );
        content.add( BorderLayout.NORTH,  new JButton( "North" ) );
        content.add( "South",  new JButton( "South" ) );
        content.add( "East",   new JButton( "East" ) );
        content.add( "West",   new JButton( "West" ) );
        content.add( "Center", new JButton( "Center" ) );

        win.setVisible(true);
    }
} // ShowBorderLayout
```

BorderLayout







Containers

- A **`JFrame`** is not the only type of container that you can use in Swing
- The subclasses of **`Container`** are:
  - JPanel
  - JWindow
  - JApplet
- **`Window`** is subclassed as follows:
  - JDialog
  - JFrame

Swing Components