Object Oriented Programming using Java

Prepared By:
Suyel, PhD
Assistant Professor
Dept. of CSE, NIT Patna

Outline

1. Basics of AWT
2. Window Fundamentals
3. Working with Frame Windows
4. Frame Class
5. Creating a Frame Window
6. Working with Graphics
7. Working with Font
8. Control Fundamentals
9. Layout Manager

Basics of AWT

- Abstract Window Toolkit (AWT) contains numerous classes and methods that allow us to create and manage windows.
- Common use of AWT is in Applets.
- It is the foundation upon which Swing is built. However, AWT is not important now.
- As Swing is built on the top of AWT, many AWT classes are used either directly or indirectly by Swing.
- **Some** of the classes of AWT are mentioned in the next 2 slides, which are in java.awt package.

Basics of AWT (Cont...)

Class	Description
AWTEvent	Encapsulates AWT events.
AWTEventMulticaster	Dispatches events to multiple listeners.
BorderLayout	The border layout manager. Border layouts use five components: North, South, East, West, and Center.
Button	Creates a push button control.
Canvas	A blank, semantics-free window.
Checkbox	Creates a check box control.
CheckboxGroup	Creates a group of check box controls.
Choice	Creates a pop-up list.
Color	Manages colors in a portable, platform-independent fashion.
Cursor	Encapsulates a bitmapped cursor.

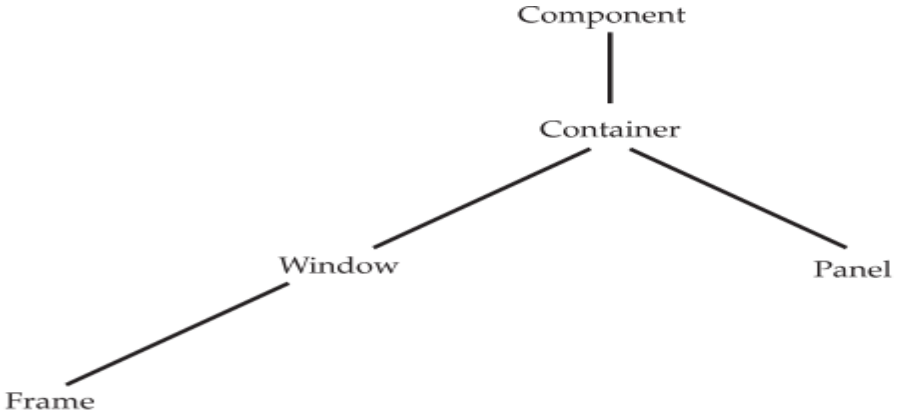
Basics of AWT (Cont...)

Class	Description
Dimension	Specifies the dimensions of an object. The width is stored in width, and the height is stored in height.
Event	Encapsulates events.
FlowLayout	Flow layout positions components left to right, top to bottom.
Font	Encapsulates a type font.
Frame	Creates a standard window that has a title bar, resize corners and a menu bar.
Graphics	Encapsulates the graphics context. This context is used by the various output methods to display output in a window.
Image	Encapsulates graphical images.
Label	Creates a label that displays a string.
Menu	Creates a pull-down menu.

Window Fundamentals

- AWT defines windows according to a class hierarchy that adds functionality and specificity with each level.
- There are two most common windows, which are derived from:

1. **Panel:** This is used by applets.
2. **Frame:** Creates a standard application window.



Window Fundamentals (Cont...)

Component

- Component is an abstract class that encapsulates all of the attributes of a visual component.
- All user interface elements that are displayed on the screen and interact with the user are subclasses of Component.
- It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning, etc.

Container

- It is a subclass of Component class.
- Container is a component in AWT that contains another component like button, text field, tables, etc. and creates nested within it.
- It keeps track of components that are added to another component.

Window Fundamentals (Cont...)

Panel

- Panel class is a concrete subclass of Container. It doesn’t add any new methods; it simply implements Container.
- A Panel is a window. It doesn't contain a title bar and menu bar.
- It is a container used for holding components like button, textfield, etc.
- Other components can be added to a Panel object by its add( ) method. Once these components are added, we can position and resize them manually by using the setLocation( ), setSize( ), etc.

Window

- Window class creates a top level window. The window is the container that does not have borders and menu bars.
- Generally, we cannot create Window objects directly. Instead, we use a subclass of Window called Frame.

Window Fundamentals (Cont...)

Frame

- Frame is a subclass of Window.
- It has a title bar, menu bar, borders, and resizing corners
- It is a container that contains several different components like button, title bar, textfield, label, etc.
- In Java, most of the AWT applications are created using Frame window.

Canvas

- Canvas is not a part of the hierarchy for applet or frame window.
- It encapsulates a blank window upon which we can draw.

Working with Frame Windows

- We can use Frame to create child windows within applets and top-level or child windows for stand-alone applications.

- Frame has two constructors:

  1. Frame( ) //Creates a standard window without a title.
  2. Frame(String *title*) //Creates a window with the specified *title*.

- We cannot specify the dimensions of the window. We must set the size of the window after it has been created.

Working with Frame Windows (Cont...)

Setting the Window's Dimensions

- The setSize( ) method is used to set the dimensions. Its signatures are:

  1. void setSize(int *newWidth*, int *newHeight*) //Window size specified by *newWidth* and *newHeight*.

  2. void setSize(Dimension *newSize*) //The width and height fields of the Dimension object passed in *newSize*.

- The getSize( ) method is used to obtain the current size of a window.

  Dimension getSize( )

Hiding and Showing a Window

- After a frame window is created, it is not visible until we call the setVisible( ). Its signature is shown here:

  void setVisible(boolean *visibleFlag*)

Working with Frame Windows (Cont...)

Setting a Window's Title

- We can change the title of a frame window by using setTitle( ).
- The general form is:

  void setTitle(String *newTitle*) //*newTitle* is the new title of the window.

Closing a Frame Window

- The program must remove the window from the screen by calling setVisible(false), when it is closed.
- To intercept a window-close event, we must implement the windowClosing( ) method of the WindowListener interface.
- Inside windowClosing( ), we must remove the window from the screen.

Frame Class

- The Frame class is a top level window.
- The type of window is defined by Frame.
- It has border and title.
- It uses BorderLayout as default layout manager.

Frame Class (Cont...)

- There are many fields of java.awt.Frame class. **Some** are:
  - **static int CROSSHAIR_CURSOR:** Deprecated. Replaced by Cursor.CROSSHAIR_CURSOR
  - **static int DEFAULT_CURSOR:** Deprecated. Replaced by Cursor.DEFAUL _CURSOR
  - **static int E_RESIZE_CURSOR:** Deprecated. Replaced by Cursor. E_RESIZE_CURSOR
  - **static int MOVE_CURSOR:** Deprecated. Replaced by Cursor. MOVE_CURSOR

Frame Class (Cont...)

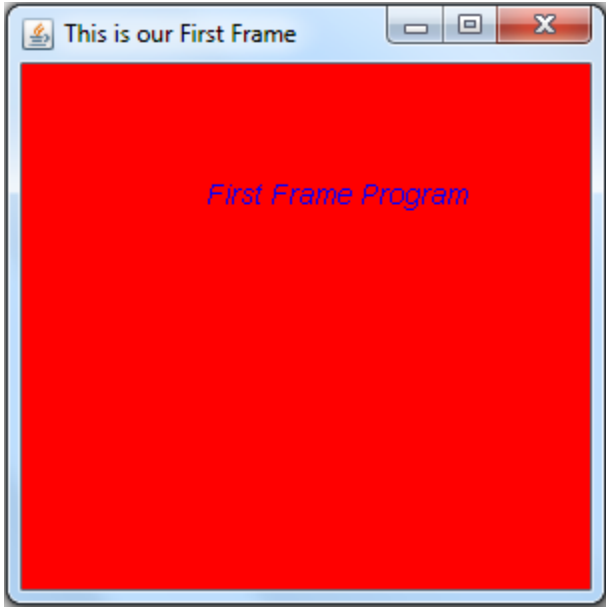
- There are many methods. Some are:
  - **addNotify():** Makes the Frame displayable by using screen recourse. Ex: void addNotify()
  - **dispose():** Disposes the Frame. Ex: void disposes()
  - **getFrames():** Returns an array of all Frames. Ex: static Frame[] getFrames()
  - **getIconImage():** Returns the image to be displayed as the icon for the Frame. Ex: Image getIconImage()
  - **getTitle():** Gets the title of the Frame. Ex: String getTitle()
  - **getMenuBar():** Gets the menu bar for the Frame. Ex: MenuBar getMenuBar()

Creating a Frame Window

```
import java.awt.*;
import java.awt.event.*;
public class Frame1 extends Frame
{
    String title;
    public Frame1(String title)
    {
      super(title);
      setSize(300, 300);
      setLayout(null);
      setVisible(true);

      addWindowListener(new WindowAdapter()
      {
        public void windowClosing(WindowEvent e)
        {
          dispose();
        }
      }
      );
    }

    public void paint(Graphics g)
    {
        setForeground(Color.BLUE);
        setBackground(Color.RED);
        Font f = new Font("Helvetica", Font.ITALIC, 14);
        g.setFont(f);
        g.drawString("First Frame Program", 100, 100);
    }

    public static void main(String args[])
    {
         new Frame1("This is our First Frame");

    }
}
```

Creating a Frame Window (Cont…)

- **Output**



Working with Graphics

- All graphics are drawn relative to a window. This can be the main window of an applet, a child window of an applet or a stand-alone application window.
- All output to a window takes place through a graphics context. A graphics context is encapsulated by the Graphics class and is obtained in two ways:
  1. By using paint( ) or update( )
  2. It is returned by the getGraphics( ) method of Component.
- The Graphics class defines a number of drawing functions. Objects are drawn and filled in the currently selected graphics color, which is black by default.
- When a graphics object is drawn that exceeds the dimensions of the window, output is automatically clipped.

Working with Graphics (Cont...)

Drawing Lines:

- Lines are drawn by using the drawLine( ) method, which are given below:

  void drawLine(int *startX*, int *startY*, int *endX*, int *endY*)

  Here, drawLine( ) displays a line in the current drawing color that begins at *startX*, *startY* and ends at *endX*, *endY*.

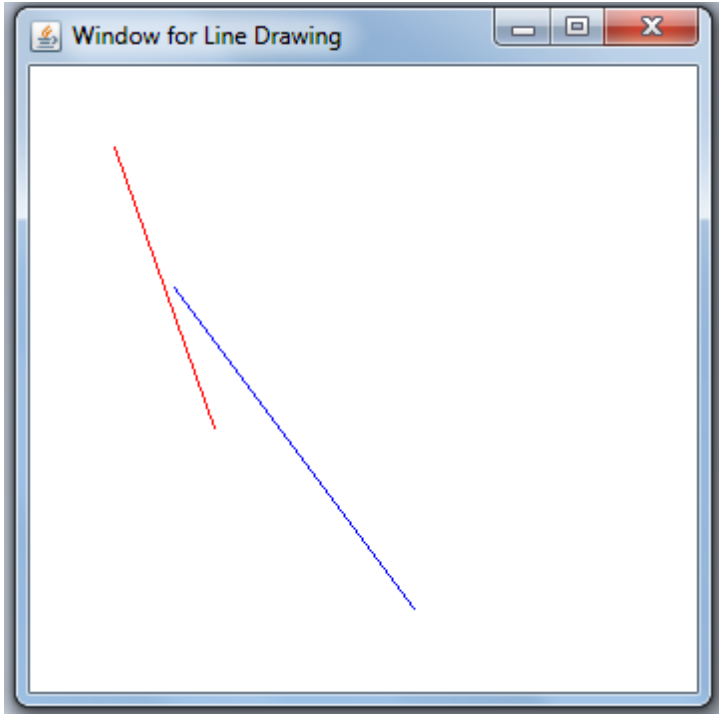
Working with Graphics (Cont...)

- **Drawing Lines (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class DrawLine1 extends Frame
{
  public DrawLine1()
  {
    setTitle("Window for Line Drawing");
    setSize(350, 350);
    setVisible(true);

    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );
  }

  public void paint(Graphics g)
  {
    g.setColor(Color.RED);
    g.drawLine(50,70, 100, 210);

    g.setColor(Color.BLUE);
    g.drawLine(80, 140, 200, 300);
  }

  public static void main(String args[])
  {
    new DrawLine1();
  }
}
```

Working with Graphics (Cont...)

- **Drawing Lines (Cont…):**
  - **Output**



Working with Graphics (Cont...)

Drawing Rectangle:

- drawRect( ) and fillRect( ) methods display an outlined and filled rectangle, respectively. They are shown here:

  void drawRect(int *top*, int *left*, int *width*, int *height*)
  void fillRect(int *top*, int *left*, int *width*, int *height*)

  Here, the upper-left corner of the rectangle is at *top*, *left*. The dimensions of the rectangle are specified by *width* and *height*.

- To draw a rounded rectangle, use drawRoundRect( ) or fillRoundRect( ).

  void drawRoundRect(int *top*, int *left*, int *width*, int *height*, int *xDiam*, int *yDiam*)
  void fillRoundRect(int *top*, int *left*, int *width*, int *height*, int *xDiam*, int *yDiam*)

  Here, the diameter of the rounding arc along the X axis is specified by *xDiam*. The diameter of the rounding arc along the Y axis is specified by *yDiam*.

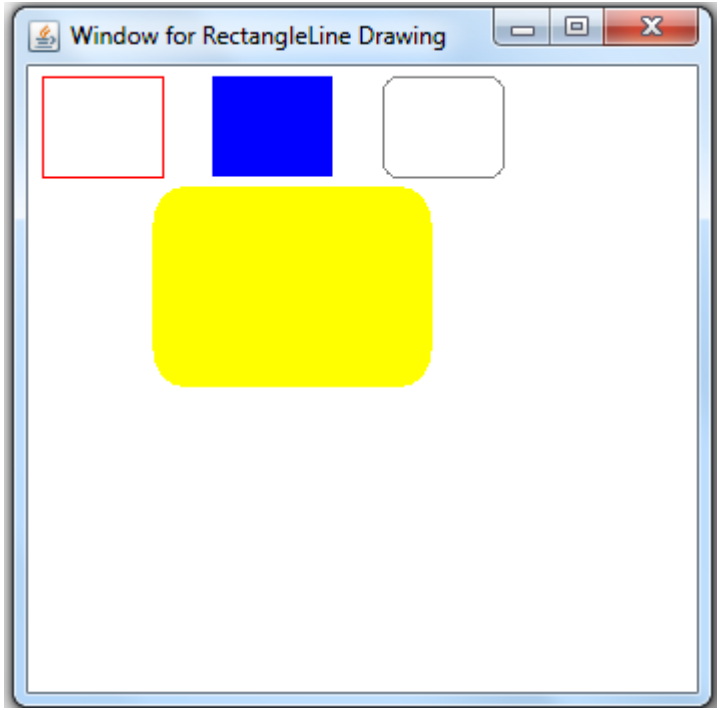
Working with Graphics (Cont...)

- **Drawing Rectangle (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class DrawRect1 extends Frame
{
  public DrawRect1()
  {
    setTitle("Window for RectangleLine Drawing");
    setSize(350, 350);
    setVisible(true);

    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );
  }

  public void paint(Graphics g)
  {
    g.setColor(Color.RED);
    g.drawRect(15, 35, 60, 50);

    g.setColor(Color.BLUE);
    g.fillRect(100, 35, 60, 50);

    g.setColor(Color.GRAY);
    g.drawRoundRect(185, 35, 60, 50, 15, 15);

    g.setColor(Color.YELLOW);
    g.fillRoundRect(70, 90, 140, 100, 30, 40);
  }

  public static void main(String args[])
  {
    new DrawRect1();
  }
}
```

Working with Graphics (Cont...)

- **Drawing Rectangle (Cont…):**
  - **Output**



Working with Graphics (Cont...)

Drawing Arc:

- Arcs can be drawn with drawArc( ) and fillArc( ) as given below:

  void drawArc(int *top*, int *left*, int *width*, int *height*, int *startAngle*, int *sweepAngle*)

  void fillArc(int *top*, int *left*, int *width*, int *height*, int *startAngle*, int *sweepAngle*)

  The arc is bounded by the rectangle. So, upper-left corner is specified by *top*, *left* and whose width and height are specified by *width* and *height*.

  The arc is drawn from *startAngle* through the angular distance specified by *sweepAngle*.

  Angles are specified in degrees. Zero degrees is on the horizontal, and it is at the 3 o'clock position. The arc is drawn counterclockwise, if *sweepAngle* is positive, and clockwise, if *sweepAngle* is negative. So, to draw an arc from 12 o'clock to 6 o'clock, the start angle would be 90 and the sweep angle 180.

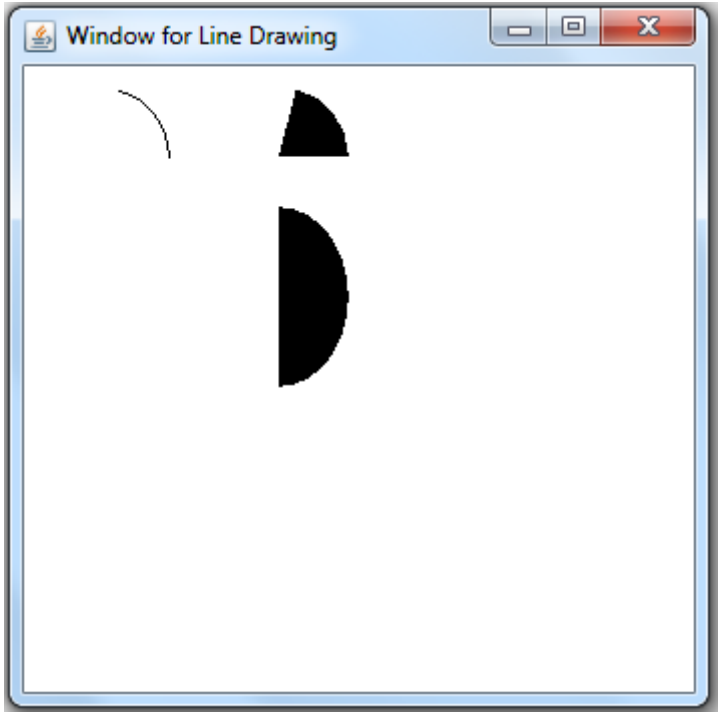
Working with Graphics (Cont...)

- **Drawing Arc (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class DrawArc1 extends Frame
{
  public DrawArc1()
  {
    setTitle("Window for Line Drawing");
    setSize(350, 350);
    setVisible(true);

    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );
  }

  public void paint(Graphics g)
  {
    g.drawArc(10, 40, 70, 70, 0, 75);
    g.fillArc(100, 40, 70, 70, 0, 75);
    g.fillArc(100, 100, 70, 90, 90, -180);
  }

  public static void main(String args[])
  {
    new DrawArc1();
  }
}
```

Working with Graphics (Cont...)

- **Drawing Arc (Cont…):**
  - **Output**



Working with Graphics (Cont...)

Working with Color:

- AWT color system allows us to specify any color that we want.
- We can also create our own colors by using one of the color constructors. Three commonly used forms are give below:

1. Color(int *red*, int *green*, int *blue*) //These values must be between 0 and 255
2. Color(int *rgbValue*) //The integer is organized with red in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7.
3. Color(float *red*, float *green*, float *blue*) //Values must be between 0.0 and 1.0

- **Using Hue, Saturation, and Brightness**

The *hue-saturation-brightness (HSB)* color model is an alternative to red-green-blue (RGB) for specifying particular colors.

The hue is specified with a number between 0.0 and 1.0. *Saturation* is another scale ranging from 0.0 to 1.0, representing light pastels to intense hues. *Brightness* values also range from 0.0 to 1.0, where 1 is bright white and 0 is black.

Working with Graphics (Cont...)

Working with Color (Cont…):

Using Hue, Saturation, and Brightness (Cont…)

Color supplies two methods that let us convert between RGB and HSB. They are shown here:

1. static int HSBtoRGB(float *hue*, float *saturation*, float *brightness*)
2. static float[ ] RGBtoHSB(int *red*, int *green*, int *blue*, float *values*[ ])

HSBtoRGB( ) returns a packed RGB value compatible with the Color(int) constructor and RGBtoHSB( ) returns a float array of HSB values corresponding to RGB integers.

If *values* is not null, then this array is given the HSB values and returned. Otherwise, a new array is created and the HSB values are returned in it.

In either case, the array contains the hue at index 0, saturation at index 1 and brightness at index 2.

We can get the red, green, and blue components of a color independently using getRed( ), getGreen( ) and getBlue( ):

int getRed( )
int getGreen( )
int getBlue( )

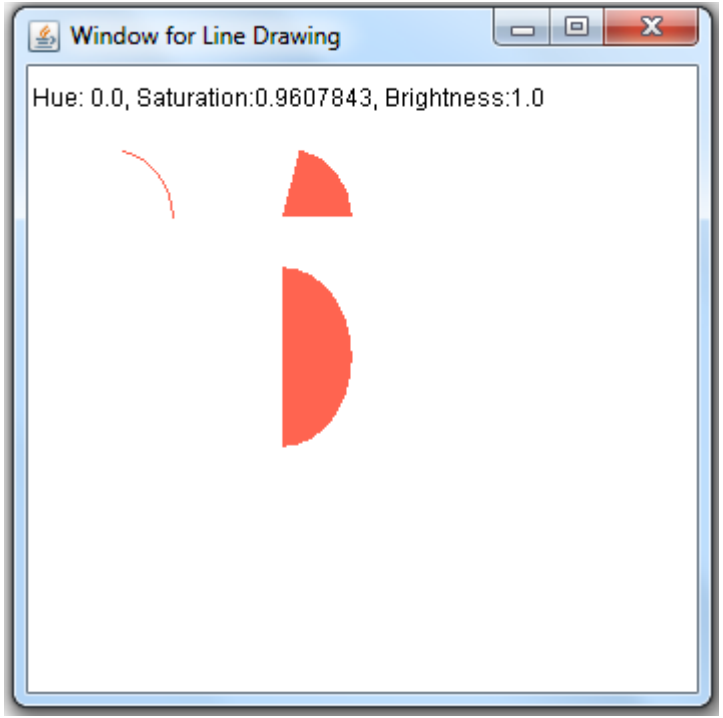
Working with Graphics (Cont...)

- **Working with Color (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class Color1 extends Frame
{
  public Color1()
  {
    setTitle("Window for Line Drawing");
    setSize(350, 350);
    setVisible(true);

    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );
  }

  public void paint(Graphics g)
  {
    float[] hsbValues = new float[3];
    hsbValues = Color.RGBtoHSB(255,10,10,hsbValues);
    float hue, saturation, brightness;
    hue = hsbValues[0];
    saturation = hsbValues[1];
    brightness = hsbValues[2];
    g.drawString("Hue: " + hue + ", Saturation:" + saturation + ",
                                  Brightness:" + brightness,10,50);

    Color c = new Color(255,100,80);
    g.setColor(c);
    g.drawArc(10, 70, 70, 70, 0, 75);
    g.fillArc(100, 70, 70, 70, 0, 75);
    g.fillArc(100, 130, 70, 90, 90, -180);
  }

  public static void main(String args[])
  {
    new Color1();
  }
}
```

Working with Graphics (Cont...)

- **Working with Color (Cont..):**
  - **Output**



Working with Graphics (Cont...)

Setting the Paint Mode:

- *Paint mode* supports a new output to a window overwrites any preexisting contents.
- However, it is possible to have new objects XORed onto the window by using setXORMode( ) as given below:

  void setXORMode(Color *xorColor*)

  Here, *xorColor* specifies the color that will be XORed to the window, when an object is drawn.
- To return to overwrite mode, we call setPaintMode( ):

  void setPaintMode( )

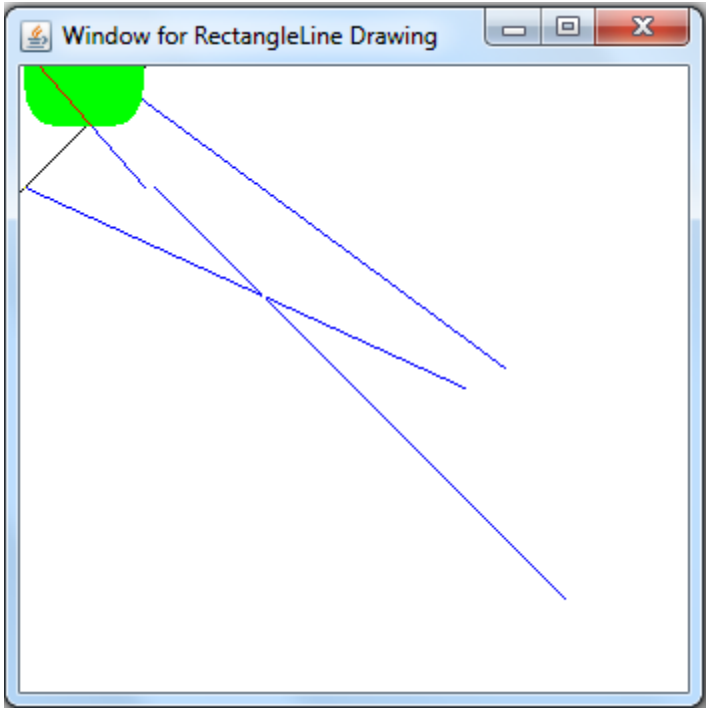
Working with Graphics (Cont...)

- **Setting the Paint Mode (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class PaintMethod1 extends Frame
{
  public PaintMethod1()
  {
    setTitle("Window for RectangleLine Drawing");
    setSize(350, 350);
    setVisible(true);

    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );
  }

  public void paint(Graphics g)
  {
    g.drawLine(0, 100, 100, 0);
    g.setColor(Color.BLUE);
    g.drawLine(40, 25, 250, 180);
    g.drawLine(75, 90, 280, 295);
    g.setColor(Color.GREEN);
    g.fillRoundRect(10, 10, 60, 50, 30, 40);

    g.setXORMode(Color.RED);
    g.drawLine(10, 90, 230, 190);
    g.drawLine(9, 20, 70, 90);
    g.setPaintMode();
  }

  public static void main(String args[])
  {
    new PaintMethod1();
  }
}
```

Working with Graphics (Cont...)

- **Setting the Paint Mode (Cont…):**
  - **Output**



Working with Font

- Fonts are encapsulated by the Font class. This class defines many methods. **Some** are listed below:

Method	Description
static Font decode(String str)	Returns a font with the given name.
int getSize( )	Returns the size (in points) of the invoking font.
int getStyle( )	Returns the style values of the invoking font.
boolean isBold( )	Returns true, if the font includes the BOLD style value. Otherwise, false is returned.

Working with Font (Cont…)

- **getAvailableFontFamilyNames( ) and getAllFonts( ):**
  - Both these methods are defined by the GraphicsEnvironment class and return all the available fonts. It is shown below:

    1. String[ ] getAvailableFontFamilyNames( ) //Return an array of string
    2. Font[ ] getAllFonts( ) //Returns an array of Font object

    As these methods are members of GraphicsEnvironment, we need a GraphicsEnvironment reference to call them. We can obtain this reference by using the getLocalGraphicsEnvironment( ) static method. It is shown here:

    static GraphicsEnvironment getLocalGraphicsEnvironment( )

    //GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();

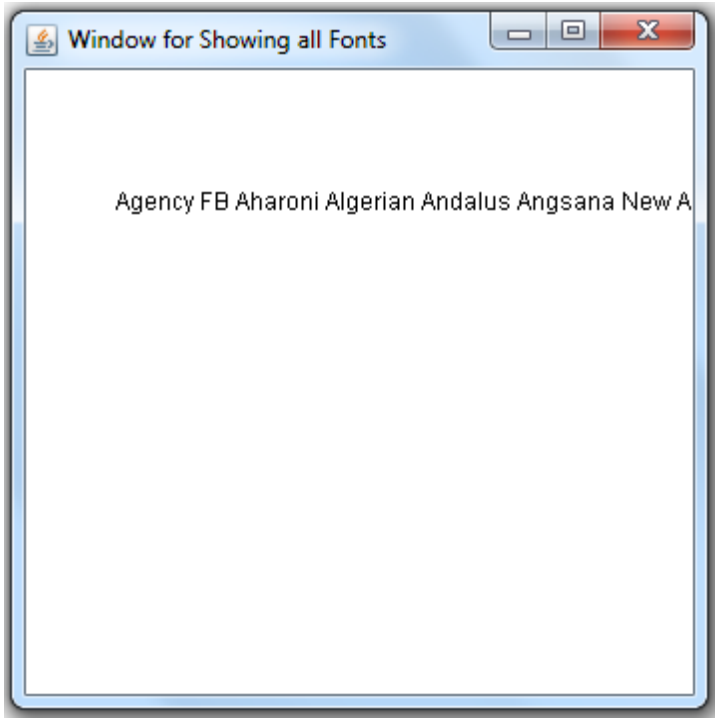
Working with Font (Cont…)

- **getAvailableFontFamilyNames( ) and getAllFonts( ) (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class ShowFont1 extends Frame
{
  public ShowFont1()
  {
    setTitle("Window for Showing all Fonts");
    setSize(350, 350);
    setLayout(null);
    setVisible(true);

    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );
  }

  public void paint(Graphics g)
  {
    String msg = " ";
    String FontList[];
    GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
    FontList = ge.getAvailableFontFamilyNames();
    for(int i = 0; i < FontList.length; i++)
      msg += FontList[i] + " ";
    g.drawString(msg, 50, 100);
  }

  public static void main(String args[])
  {
    new ShowFont1();

  }
}
```

Working with Font (Cont…)

- **getAvailableFontFamilyNames( ) and getAllFonts( ) (Cont…):**
  - **Output**



Control Fundamentals

- AWT supports many types of controls, such as labels, push buttons, check boxes, lists, etc.
- To include a control in a window, we must add it to the window.
- To do this, we must first create an instance of the desired control, and then, need to add it to a window by calling add( ), which is defined by Container.
- We can remove a control from a window, by calling remove( ). This method is also defined by Container.
- Except for labels, all controls generate events, when they are accessed by the user.

Control Fundamentals (Cont...)

- **Label:**
  - A *label* is an object of type Label, and it contains a string.
  - Labels are passive controls that do not support any interaction with the users. Label defines the following constructors:

    1. Label( ) throws HeadlessException //Creates a blank label
    2. Label(String *str*) throws HeadlessException //Left justified string *str*
    3. Label(String *str*, int *how*) throws HeadlessException

    The last version contains the string specified by str using the alignment specified by *how*. The value of *how* must be: Label.LEFT, Label.RIGHT or Label.CENTER.

    The HeadlessException can be used to write code that can adapt to non-interactive environments.

Control Fundamentals (Cont...)

Label (Cont…):

- We can set or change the text in a label by using setText( ). We can obtain the current label by calling getText( ). These methods are:

  1. void setText(String *str*)
  2. String getText( )

- We can set the alignment of the string within the label by calling setAlignment( ). To obtain the current alignment, we can call getAlignment( ). The methods are as follows:

  1.void setAlignment(int *how*)
  2. int getAlignment( )

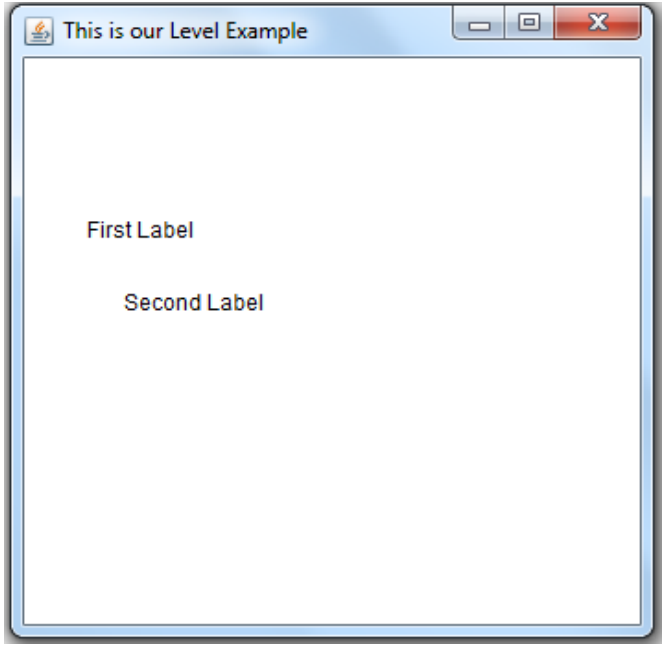
Control Fundamentals (Cont...)

- **Label (Cont…)**

```
import java.awt.*;
class Label1
{
  public static void main(String args[])
  {
    Frame f = new Frame("This is our Level Example");
    f.setSize(350, 350);
    f.setLayout(null);
    f.setVisible(true);

    Label l1,l2;
    l1 = new Label("First Label");
    l1.setBounds(40, 100, 100, 50);
    l2 = new Label("Second Label", Label.RIGHT);
    l2.setBounds(40, 140, 100, 50);
    f.add(l1);
    f.add(l2);
  }
}
```

Control Fundamentals (Cont...)

- **Label (Cont…)**
  - **Output**



Control Fundamentals (Cont...)

Push Button:

- A *Push button* is a component that contains a label and generates an event, when it is pressed.
- Push buttons are objects of type Button. Button defines below mentioned two constructors:

1. Button( ) throws HeadlessException //Creates an empty button
2. Button(String *str*) throws HeadlessException //Contains *str* as a label

- After a button has been created, we can set its label by calling setLabel( ). We can retrieve its label by calling getLabel( ). These methods are as follows:

1. void setLabel(String *str*) //*str* is the new label for the button
2. String getLabel( )

Control Fundamentals (Cont...)

Push Button (Cont…):

- Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered.
- Each listener implements the ActionListener interface. That interface defines the actionPerformed( ) method, which is called when an event occurs.
- An ActionEvent object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the *action command string* associated with the button.
- By default, the action command string is the label of the button. Usually, either the button reference or the action command string can be used to identify the button.

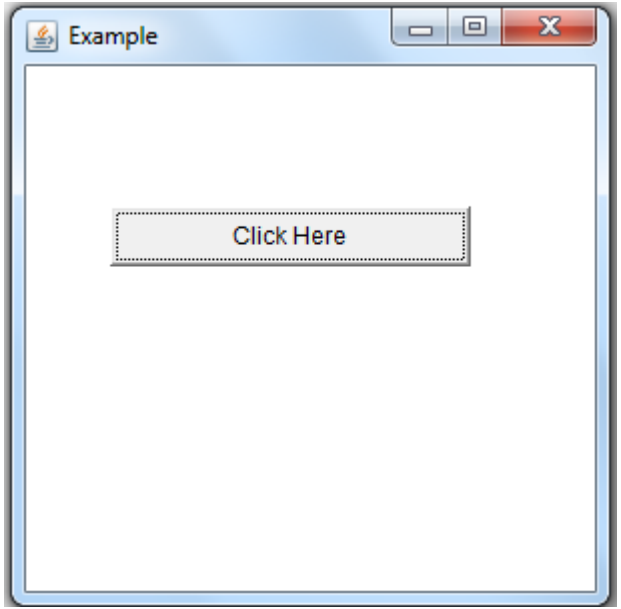
Control Fundamentals (Cont...)

- **Push Button (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class Button2 extends Frame
{
    public static void main(String args[])
    {
      Frame f = new Frame ("Example");
      f.setSize(300, 300);
      f.setLayout(null);
      f.setVisible(true);

      Button b=new Button("Click Here");
      b.setBounds(50,100,180,30);
      f.add(b);

      f.addWindowListener(new WindowAdapter()
      {
        public void windowClosing(WindowEvent e)
        {
          f.dispose();
        }
      }
      );

    }
}
```

Control Fundamentals (Cont...)

- **Push Button (Cont…):**
  - **Output**



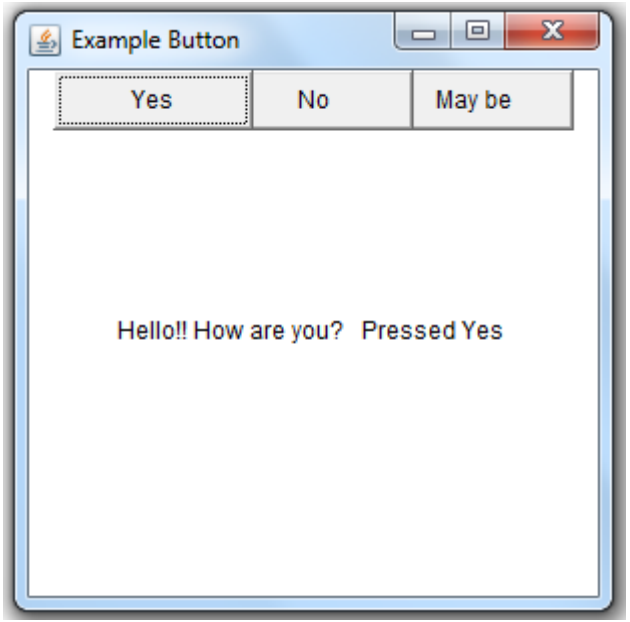
Control Fundamentals (Cont...)

- **Push Button (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class Button3 extends Frame implements ActionListener
{
    Button yes, no, maybe; Label l;
    public Button3()
    {
      setTitle("Example Button");
      l=new Label();
      l.setBounds(50, 150, 250, 20);
      yes = new Button("Yes");
      yes.setBounds(20, 30, 100, 30);
      no = new Button("No");
      no.setBounds(100, 30, 100, 30);
      maybe = new Button("May be");
      maybe.setBounds(180, 30, 100, 30);
      add(yes); add(no); add(maybe); add(l);
      yes.addActionListener(this);
      no.addActionListener(this);
      maybe.addActionListener(this);
      setSize(300, 300);   setLayout(null);    setVisible(true);
      addWindowListener(new WindowAdapter()
      {
        public void windowClosing(WindowEvent e)
        {
          dispose();
        }
      }
      );
    }
    public void actionPerformed(ActionEvent ae)
    {
      String str = ae.getActionCommand();
      if(str.equals("Yes"))
      {
        l.setText("Hello!! How are you?    Pressed Yes");
      }
      else if(str.equals("No"))
      {
        l.setText("I am fine.    pressed No");
      }
      else
      {
        l.setText("Will you play Cricket?    Pressed May Be");
      }
    }
    public static void main(String args[])
    {
      new Button3();
    }
}
```

Control Fundamentals (Cont...)

- **Push Button (Cont…):**
  - **Output**



  - **When we will press different Button, different Text will be shown based on Code.**

Control Fundamentals (Cont...)

TextField:

- TextField class implements a single-line text-entry area, usually called an *edit control.*
- TextField is a subclass of TextComponent. It defines the following constructors:

1. TextField( ) throws HeadlessException //Default text field
2. TextField(int *numChars*) throws HeadlessException //Create with *numChars* width
3. TextField(String *str*) throws HeadlessException //Create a text field with the *str*
4. TextField(String *str*, int *numChars*) throws HeadlessException

- To obtain the string currently contained in the text field, we call getText( ). To set the text, we call setText( ). These methods are as follows:

1. String getText( )
2. void setText(String *str*)

Control Fundamentals (Cont...)

TextField (Cont…):

- We can select a portion of text by using select( ). The program can obtain the currently selected text by calling getSelectedText( ).

```
1. getSelectedText( )                          //Returns the selected text
2. void select(int startIndex, int endIndex)   //0 to n-1
```

- We can control whether the contents of a text field can be modified by calling setEditable( ). We can determine editability by calling isEditable( ).

```
1. boolean isEditable( )                  //Returns true, if text can be modified
2. void setEditable(boolean canEdit)      //If canEdit is true, the text may be changed
```

Control Fundamentals (Cont...)

- **TextField (Cont…):**
  - There may be possibility that we want to enter text that is not displayed.
  - We can disable the echoing of the characters as they are typed by calling setEchoChar( ). This method specifies a single character that the TextField displays, when characters are entered.
  - We can check a text field to see, if it is in the echoing mode with the echoCharIsSet( ).
  - We can retrieve the echo character by calling getEchoChar( ). These methods are shown below:

    1. void setEchoChar(char *ch*)
    2. boolean echoCharIsSet( )
    3. char getEchoChar( )

Control Fundamentals (Cont...)

- **TextField (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class TextField1 extends Frame
{
  String title;
  public TextField1(String title)
  {
    super(title);
    TextField t1,t2;
    t1=new TextField("Welcome to First TextField 1");
    t1.setBounds(30, 100, 250, 40);
    t2=new TextField("Welcome to Second TextField 2");
    t2.setBounds(30, 170, 250, 40);
    add(t1);
    add(t2);
    setSize(300, 300);
    setLayout(null);
    setVisible(true);

    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );
  }

  public static void main(String args[])
  {
    new TextField1 ("This is our FIrst Text Field Example");
  }
}
```

Control Fundamentals (Cont...)

- **TextField (Cont…):**
  - **Output**

Control Fundamentals (Cont...)

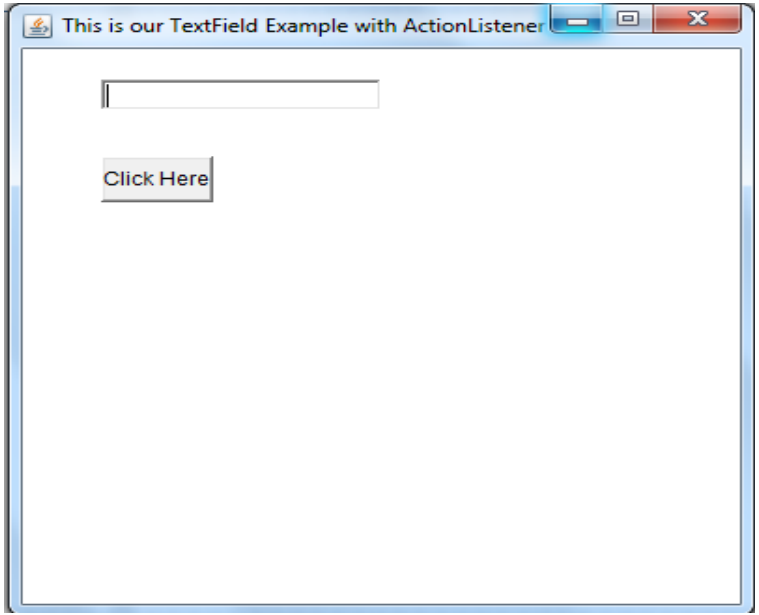
- **TextField (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class Textfield2 extends Frame implements ActionListener
{
  TextField tf; Button b; Label l;
  public Textfield2()
  {
    setTitle("This is our TextField Example with ActionListener");
    tf=new TextField();
    tf.setBounds(50, 50, 150, 20);
    l=new Label();
    l.setBounds(50, 150, 250, 20);
    b=new Button("Click Here");
    b.setBounds(50, 100, 60, 30);
    b.addActionListener(this);
    add(tf);add(b);add(l);
    setSize(400,400);
    setLayout(null);
    setVisible(true);

    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );
  }

  public void actionPerformed(ActionEvent e)
  {
    String s1= tf.getText();
    String s2="After Clicking Button";
    String s3=s1+"+"+s2;
    l.setText(s3);
  }

  public static void main(String args[])
  {
    new Textfield2 ();
  }
}
```

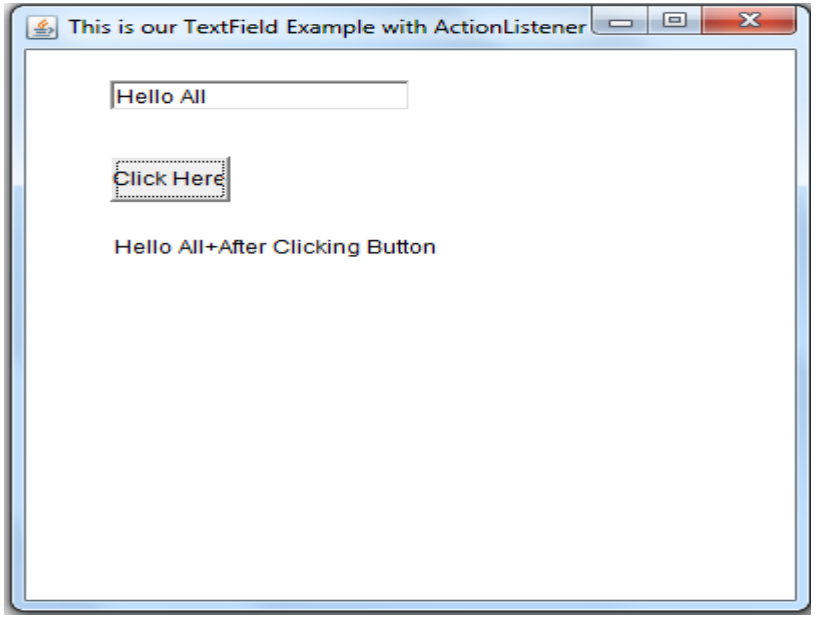
Control Fundamentals (Cont...)

- **TextField (Cont…):**
  - **Output**

Before Clicking Button



After Clicking Button



Control Fundamentals (Cont...)

- **TextArea:**
  - The object of a TextArea class is a multi line region that displays text.
  - It allows to edit multiple line text.
  - The following are the constructors of TextArea:

    1. TextArea( ) throws HeadlessException
    2. TextArea(int *numLines*, int *numChars*) throws HeadlessException
    3. TextArea(String *str*) throws HeadlessException
    4. TextArea(String *str*, int *numLines*, int *numChars*) throws HeadlessException
    5. TextArea(String *str*, int *numLines*, int *numChars*, int *sBars*) throws HeadlessException

    Here, *numLines* specifies the height in lines. *numChars* specifies its width in characters.

    In the fifth form, we can specify the scroll bars that we want the control to have. *sBars* must be one of these values: SCROLLBARS_BOTH, SCROLLBARS_NONE, SCROLLBARS_HORIZONTAL_ONLY, SCROLLBARS_VERTICAL_ONLY.

Control Fundamentals (Cont...)

TextArea (Cont…):

- TextArea is a subclass of TextComponent. So, it supports the getText( ), setText( ), getSelectedText( ), select( ), isEditable( ) and setEditable( ).

- In addition, TextArea adds the following methods:

1. void append(String *str*) //Append *str* to the end of the current text
2. void insert(String *str*, int *index*) //Insert *str* at the specified index
3. void replaceRange(String *str*, int *startIndex*, int *endIndex*) //Replace the text

- Text area only generates got-focus and lost-focus events.

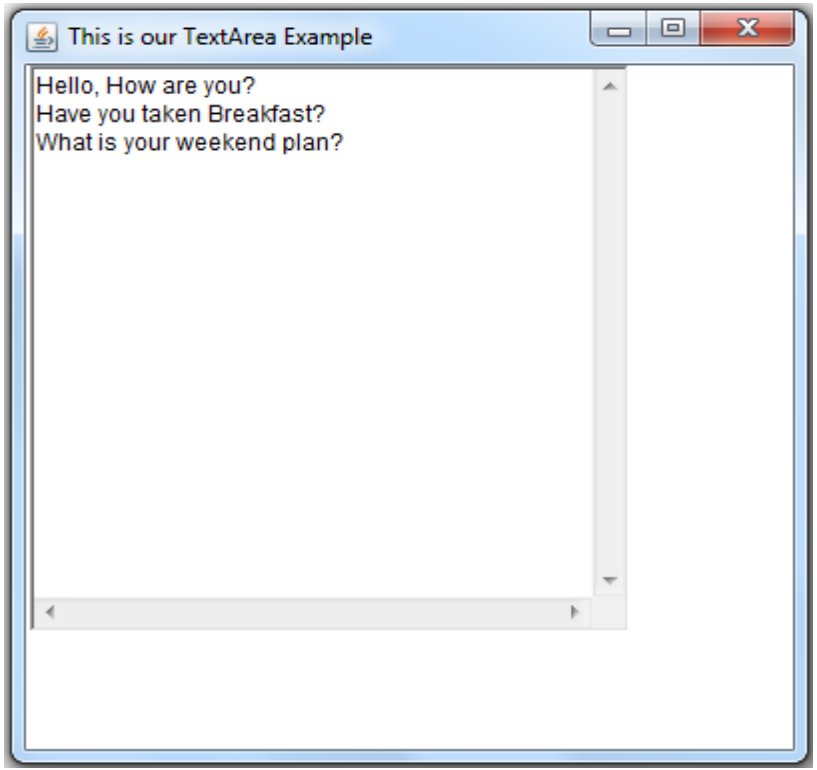
Control Fundamentals (Cont...)

- **TextArea (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class Textarea1 extends Frame
{
  public Textarea1()
  {
    setTitle("This is our TextArea Example");
    TextArea area=new TextArea("Hello, How are you?\n"+
                               "Have you taken Breakfast?\n"+
                               "What is your weekend plan?");
    area.setBounds(10,30, 300,300);
    add(area);
    setSize(400,400);
    setLayout(null);
    setVisible(true);
    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );
  }

  public static void main(String args[])
  {
    new Textarea1 ();
  }
}
```

Control Fundamentals (Cont...)

- **TextArea (Cont…):**
  - **Output**



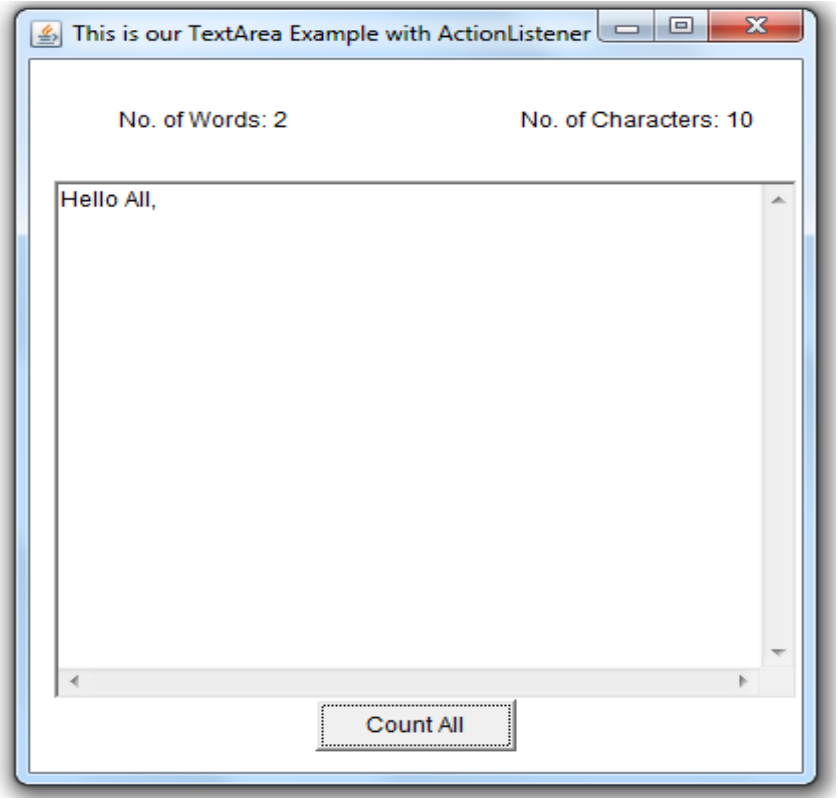
Control Fundamentals (Cont...)

- **TextArea (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class Textarea2 extends Frame implements ActionListener
{
  Label l1, l2; Button b; TextArea area;
  public Textarea2()
  {
    setTitle("This is our TextArea Example with ActionListener");

    area=new TextArea();
    area.setBounds(20, 100, 370, 300);
    l1=new Label();
    l1.setBounds(50, 50, 100, 30);
    l2=new Label();
    l2.setBounds(250, 50, 300, 30);
    b=new Button("Count All");
    b.setBounds(150, 400, 100, 30);
    b.addActionListener(this);
    add(area); add(l1); add(l2); add(b);
    setSize(400, 450);
    setLayout(null);
    setVisible(true);
    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );
  }
  public void actionPerformed(ActionEvent e)
  {
    String st=area.getText();
    String word[]=st.split("\\s");
    l1.setText("No. of Words: " + word.length);
    l2.setText("No. of Characters: " + st.length());
  }

  public static void main(String args[])
  {
    new Textarea2();
  }
}
```

Control Fundamentals (Cont...)

- **TextArea (Cont…):**
  - **Output**



Control Fundamentals (Cont...)

Check Box:

- A *check box* is a control that is used to turn an option on or off.
- There is a label associated with each check box that describes what option the box represents.
- Check boxes can be used individually or as a part of a group.
- Check boxes are objects of the Checkbox class. It supports the below mentioned five constructors:

```
Checkbox( ) throws HeadlessException                          //Create a checkbox with blank label
Checkbox(String str) throws HeadlessException                 //Create a checkbox with a label str
Checkbox(String str, boolean on) throws HeadlessException     //Create with initial state
Checkbox(String str, boolean on, CheckboxGroup cbGroup) throws HeadlessException
Checkbox(String str, CheckboxGroup cbGroup, boolean on) throws HeadlessException
```

The last two create a check box, whose label is specified by *str* and whose group is specified by *cbGroup*. If this check box is not part of a group, then *cbGroup* must be null.

Control Fundamentals (Cont...)

Check Box (Cont…):

- To retrieve the current state of a check box, we call getState( ). To set its state, we call setState( ).
- We can obtain the current label associated with a check box by calling getLabel( ). To set the label, we call setLabel( ). These methods are:

  1. boolean getState( )
  2. void setState(boolean *on*)
  3. String getLabel( )
  4. void setLabel(String *str*)

- Each time a check box is selected or deselected, an item event is generated.
- Each listener implements the ItemListener interface. That interface defines the itemStateChanged( ) method. An ItemEvent object is supplied as the argument to this method.

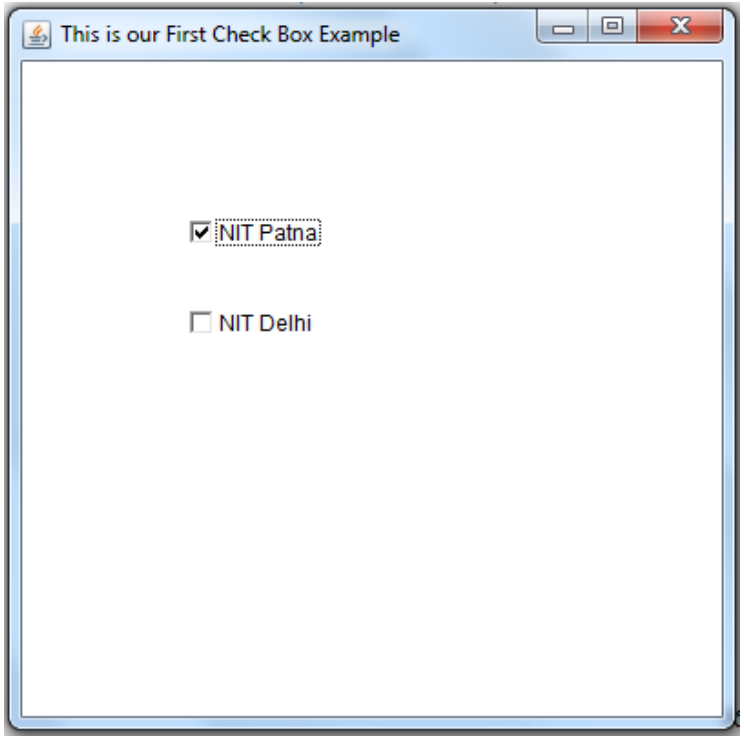
Control Fundamentals (Cont...)

- **Check Box (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class Checkbox1 extends Frame
{
  public Checkbox1()
  {
    setTitle("This is our First Check Box Example");

    Checkbox checkbox1 = new Checkbox("NIT Patna", true);
    checkbox1.setBounds(100, 100, 100, 50);
    Checkbox checkbox2 = new Checkbox("NIT Delhi");
    checkbox2.setBounds(100, 150, 100, 50);
    add(checkbox1); add(checkbox2);
    setSize(400,400);
    setLayout(null);
    setVisible(true);

    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );
  }

  public static void main(String args[])
  {
    new Checkbox1();
  }
}
```

Control Fundamentals (Cont...)

- **Check Box (Cont…):**
  - **Output**



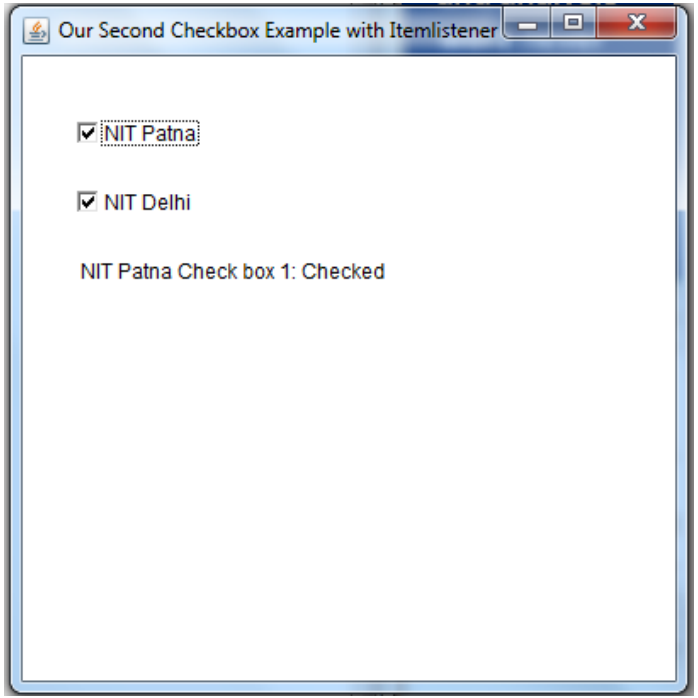
Control Fundamentals (Cont...)

- **Check Box (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class Checkbox2 extends Frame
{
  public Checkbox2()
  {
    setTitle("Our Second Checkbox Example with Itemlistener");
    Label l = new Label();
    l.setBounds(40, 130, 200, 50);
    Checkbox box1 = new Checkbox("NIT Patna");
    box1.setBounds(40, 50, 100, 50);
    Checkbox box2 = new Checkbox("NIT Delhi");
    box2.setBounds(40, 90, 100, 50);
    add(l); add(box1); add(box2);
    setSize(400, 400);      setLayout(null);       setVisible(true);
    box1.addItemListener(new ItemListener()
    {
      public void itemStateChanged(ItemEvent e)
      {
        l.setText("NIT Patna Check box 1: " + (e.getStateChange()==1?"Checked":"Unchecked"));
      }
    });
    box2.addItemListener(new ItemListener()
    {
      public void itemStateChanged(ItemEvent e)
      {
        l.setText("NIT Delhi Check box 2: " + (e.getStateChange()==1?"Checked":"Unchecked"));
      }
    });

    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );
  }
  public static void main(String args[])
  {
    new Checkbox2 ();
  }
}
```

Control Fundamentals (Cont...)

- **Check Box (Cont…):**
  - **Output**



  - **Recent Check box details will be in Label.**

Control Fundamentals (Cont...)

Check Box (Cont…):

- It is possible to create a set of mutually exclusive check boxes in which one check box in a group can be checked at once.
- These check boxes are often called ***radio buttons.***
- To create a set of mutually exclusive check boxes, we must first define the group to which they will belong, and then, specify that group, when we construct the check boxes.
- Check box groups are objects of type CheckboxGroup.
- We can determine, which check box in a group is currently selected by calling getSelectedCheckbox( ). We can set a check box by calling setSelectedCheckbox( ). These methods are:

1. Checkbox getSelectedCheckbox( )
2. void setSelectedCheckbox(Checkbox *which*)

Here, *which* is the check box that we want to be selected. The previously selected check box will be turned off.

Control Fundamentals (Cont...)

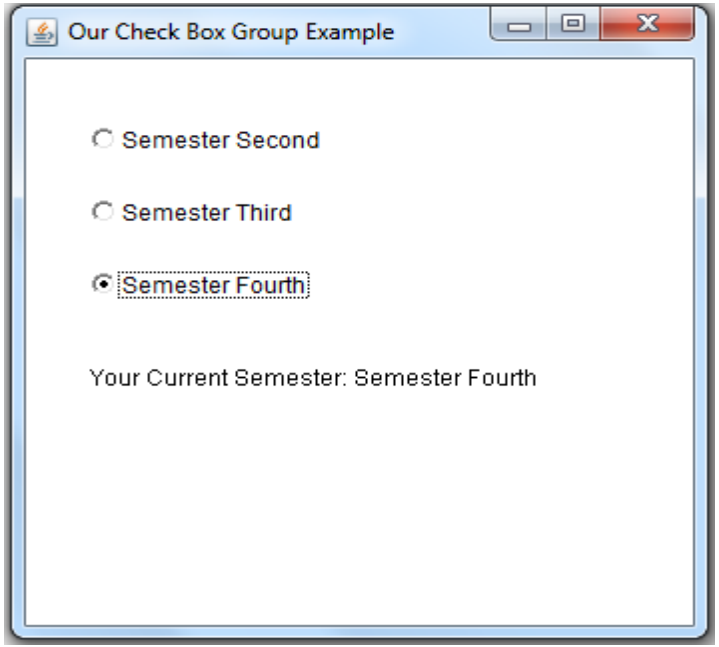
- **Check Box (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class Checkbox3 extends Frame implements ItemListener
{
  String msg = " ";
  Checkbox II, III, IV;
  CheckboxGroup cbg;
  public Checkbox3()
  {
    setTitle("Our Check Box Group Example");
    cbg = new CheckboxGroup();
    II = new Checkbox("Semester Second", cbg, true);
    II.setBounds(40, 50, 150, 50);
    III = new Checkbox("Semester Third", cbg, false);
    III.setBounds(40, 90, 150, 50);
    IV = new Checkbox("Semester Fourth", cbg, false);
    IV.setBounds(40, 130, 150, 50);
    add(II); add(III); add(IV);
    II.addItemListener(this);
    III.addItemListener(this);
    IV.addItemListener(this);
    setSize(350, 350);  setLayout(null);  setVisible(true);
    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );
  }
  public void itemStateChanged(ItemEvent ie)
  {
    repaint();
  }
  public void paint (Graphics g)
  {
    msg = "Your Current Semester: ";
    msg += cbg.getSelectedCheckbox().getLabel();
    g.drawString(msg, 40, 210);
  }
  public static void main(String args[])
  {
    new Checkbox3();
  }
}
```

Control Fundamentals (Cont...)

Check Box (Cont…):

- Output



- Based on the selection of Checkbox, Current Semester is changed.

Control Fundamentals (Cont...)

Choice:

- The Choice class is used to create a *pop-up list* of items from which the user may choose. Thus, a Choice control is a form of menu.
- When inactive, a Choice component takes up only enough space to show the currently selected item.
- Each item in the list is a string that appears as a left-justified label in the order it is added to the Choice object.
- Choice defines the default constructor i.e. Choice(), which creates an empty list.
- To add a selection to the list, we call add( ). It has this general form:

  void add(String *name*) //*name* is the name of the item being added

- To determine which item is currently selected, we can call either getSelectedItem( ) or getSelectedIndex( ). These methods are:

  1. String getSelectedItem( ) //Returns a String i.e. name of the item
  2. int getSelectedIndex( ) //The first item is selected by default and its index is 0

Control Fundamentals (Cont...)

Choice (Cont…):

- To obtain the number of items in the list, we call getItemCount( ). We can set the currently selected item using select( ) method with either a zero-based integer index or a string that matches a name in the list.

  1. int getItemCount( )
  2. void select(int *index*)
  3. void select(String *name*)

- Given an index, we can obtain the name associated with the item at that index by getItem( ), which has this general form:

  String getItem(int *index*)

- When a choice is selected, an item event is generated.
- Each listener implements the ItemListener interface. That interface defines the itemStateChanged( ) method.
- An ItemEvent object is supplied as the argument to itemStateChanged().

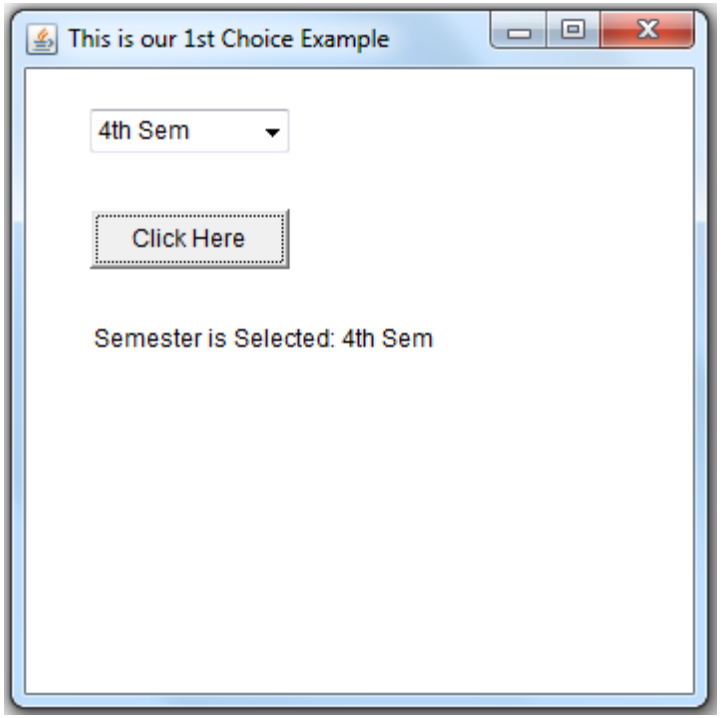
Control Fundamentals (Cont...)

- **Choice (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class Choice1 extends Frame implements ActionListener
{
  Choice c; Button b; Label l;
  public Choice1()
  {
    setTitle("This is our 1st Choice Example");
    l= new Label();
    l.setBounds(40, 150, 200, 30);
    b=new Button("Click Here");
    b.setBounds(40, 100, 100, 30);
    b.addActionListener(this);
    c=new Choice();
    c.setBounds(40, 50, 100, 30);
    c.add("2nd Sem");
    c.add("3rd Sem");
    c.add("4th Sem");
    add(c); add(b); add(l);
    setSize(350,350);
    setLayout(null);
    setVisible(true);
    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );
  }

  public void actionPerformed(ActionEvent e)
  {
    String st = "Semester is Selected: " + c.getItem(c.getSelectedIndex());
    l.setText(st);
  }

  public static void main(String args[])
  {
    new Choice1();
  }
}
```

Control Fundamentals (Cont...)

- **Choice (Cont…):**
  - **Output**



  - **Based on the selection of Choice, text of Label is changed.**

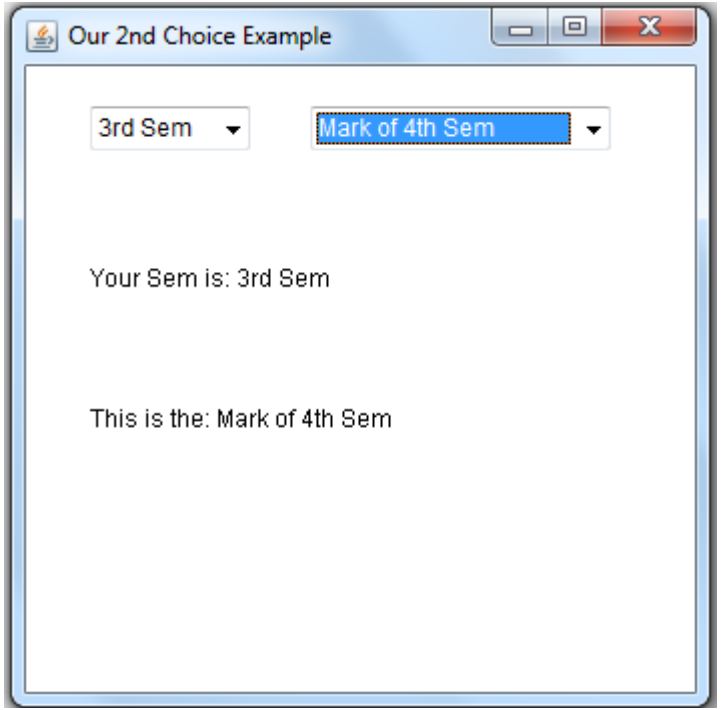
Control Fundamentals (Cont...)

- **Choice (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class Choice2 extends Frame implements ItemListener
{
  String msg = " ";
  Choice c1, c2;
  public Choice2()
  {
    setTitle("Our 2nd Choice Example");
    c1= new Choice();
    c1.setBounds(40, 50, 80, 30);
    c1.add("2nd Sem");
    c1.add("3rd Sem");
    c1.add("4th Sem");
    c2=new Choice();
    c2.setBounds(150, 50, 150, 30);
    c2.add("Mark of 2nd Sem");
    c2.add("Mark of 3rd Sem");
    c2.add("Mark of 4th Sem");
    add(c1); add(c2);
    c1.addItemListener(this);
    c2.addItemListener(this);
    setSize(350, 350); setLayout(null); setVisible(true);
    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );
  }
  public void itemStateChanged(ItemEvent ie)
  {
    repaint();
  }
  public void paint (Graphics g)
  {
    msg = "Your Sem is: ";
    msg += c1.getSelectedItem();
    g.drawString(msg, 40, 140);
    msg = "This is the: ";
    msg += c2.getSelectedItem();
    g.drawString(msg, 40, 210);
  }
  public static void main(String args[])
  {
    new Choice2();
  }
}
```

Control Fundamentals (Cont...)

- **Choice (Cont…):**
  - **Output**



  - **Based on the selection of Choice, output is changed.**

Control Fundamentals (Cont...)

- **List:**
  - List class provides a compact, multiple-choice and scrolling selection list.
  - Unlike the Choice object, which shows only the single selected item in the menu, a List object can be constructed to show any number of choices in the visible window.
  - It can also be created to allow multiple selections.
  - List provides these below mentioned constructors:

    1. List( ) throws HeadlessException //Allows to select one item at a time
    2. List(int *numRows*) throws HeadlessException //*numRows* is the number of entries that are visible
    3. List(int *numRows*, boolean *multipleSelect*) throws HeadlessException //We can select many items, if true. Otherwise, one item can be selected.

Control Fundamentals (Cont...)

List (Cont…):

- To add a selection to the list, we call add( ). It has the following two forms:

```
1. void add(String name)              //name is the item added to the end of list
2. void add(String name, int index)   //name is the item added at index
```

- For lists that allow only single selection, we can determine which item is currently selected by calling either getSelectedItem( ) or getSelectedIndex( ).

```
1. String getSelectedItem( )          //Returns a string of the item
2. int getSelectedIndex( )            //Returns the index of the item
```

- For lists that allow multiple selection to obtain the current selected items, we use either getSelectedItems( ) or getSelectedIndexes( ).

```
1. String[ ] getSelectedItems( )      //Returns an array of selected items
2. int[ ] getSelectedIndexes( )       //Returns an array of indexes of selected items
```

Control Fundamentals (Cont...)

List (Cont…):

- To obtain the number of items in the list, we call getItemCount( ).
- We can set the currently selected item by using the select( ) method with a zero-based integer index.
- These above mentioned methods are shown here:

  1. int getItemCount( )
  2. void select(int *index*)

- Given an index, we can obtain the name associated with the item at that index by calling getItem( ). Its general form is given below:

  String getItem(int *index*)

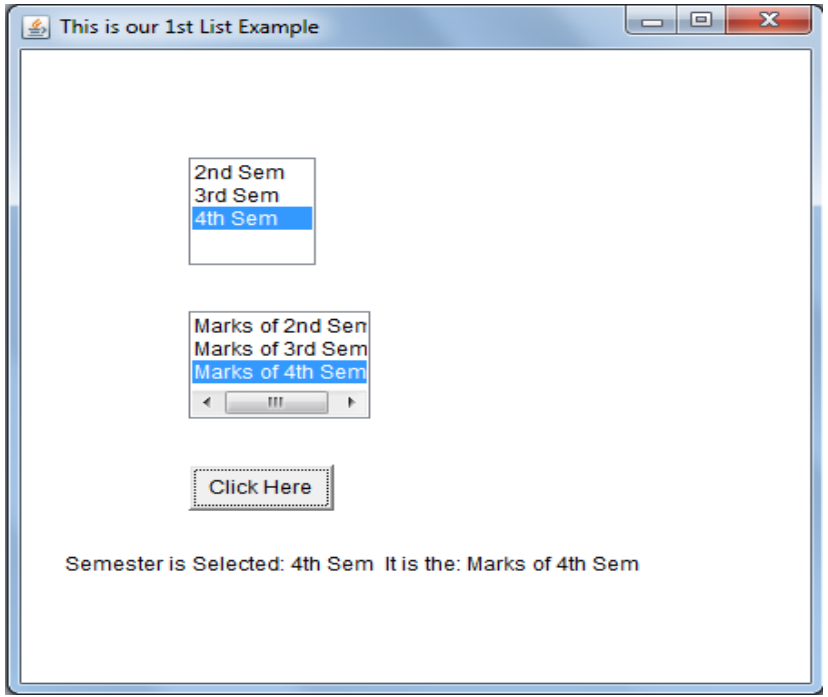
Control Fundamentals (Cont...)

- **List (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class List1
{
  List1()
  {
    Frame f=new Frame();
    f.setTitle("This is our 1st List Example");
    final Label l= new Label();
    l.setAlignment(Label.LEFT);
    l.setBounds(30, 350, 700, 30);
    Button b=new Button("Click Here");
    b.setBounds(100, 300, 80, 30);
    final List ls1=new List(3, false);
    ls1.setBounds(100, 100, 70, 70);
    ls1.add("2nd Sem");
    ls1.add("3rd Sem");
    ls1.add("4th Sem");
    final List ls2=new List(3, true);
    ls2.setBounds(100, 200, 100, 70);
    ls2.add("Marks of 2nd Sem");
    ls2.add("Marks of 3rd Sem");
    ls2.add("Marks of 4th Sem");       f.add(b);     f.add(ls1);      f.add(ls2);     f.add(l);
    f.setSize(450, 450);         f.setLayout(null);      f.setVisible(true);
    f.addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        f.dispose();
      }
    }
    );
    b.addActionListener(new ActionListener()
    {
      public void actionPerformed(ActionEvent e)
      {
        String st = "Semester is Selected: " + ls1.getItem(ls1.getSelectedIndex());
        st +="  It is the: ";
        for(String i:ls2.getSelectedItems())
        {
          st += i + " ";
        }
        l.setText(st);
      }
    });
  }
  public static void main(String args[])
  {
    new List1();
  }
}
```

Control Fundamentals (Cont...)

- **List (Cont…):**
  - **Output:**



  - **Based on the selection, contents of Label is changed.**

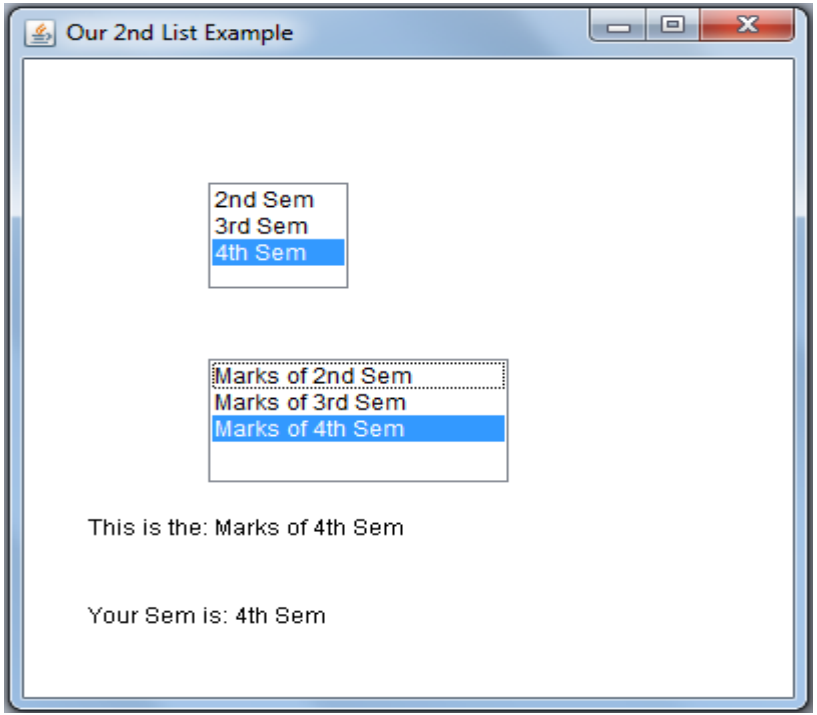
Control Fundamentals (Cont...)

- **List (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class List2 extends Frame implements ActionListener
{
  String msg = " ";
  List ls1, ls2;
  public List2()
  {
    setTitle("Our 2nd List Example");
    ls1=new List(3, false);
    ls1.setBounds(100, 100, 70, 60);
    ls1.add("2nd Sem");
    ls1.add("3rd Sem");
    ls1.add("4th Sem");
    ls2=new List(3, true);
    ls2.setBounds(100, 200, 150, 70);
    ls2.add("Marks of 2nd Sem");
    ls2.add("Marks of 3rd Sem");
    ls2.add("Marks of 4th Sem");
    add(ls1); add(ls2);
    ls1.addActionListener(this);   ls2.addActionListener(this);
    setSize(400, 400);   setLayout(null);   setVisible(true);
    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );
  }
  public void actionPerformed(ActionEvent ae)  {
    repaint();
  }
  public void paint (Graphics g)   {
    int i, ary[];
    msg = "This is the: ";
    ary=ls2.getSelectedIndexes();
    for(i=0; i<ary.length; i++)
    {
      msg += ls2.getItem(ary[i]) + " ";
    }
    g.drawString(msg, 40, 300);
    msg = "Your Sem is: ";
    msg += ls1.getSelectedItem();
    g.drawString(msg, 40, 350);
  }
  public static void main(String args[])  |{
    new List2();
  }
}
```

Control Fundamentals (Cont...)

- **List (Cont…):**
  - **Output:**



  - **Based on the selection, output is changed.**

Control Fundamentals (Cont...)

Scroll Bar:

- *Scroll bar* is used to select continuous value between a specified minimum and maximum.
- Scroll bars may be oriented horizontally or vertically.
- Here, each end has an arrow that we can click to move the current value of the scroll bar one unit in the direction of the arrow.
- The current value of the scroll bar relative to its minimum and maximum values is indicated by the *slider box* (or *thumb*) for the scroll bar.
- Scroll bars are encapsulated by the Scrollbar class.

Control Fundamentals (Cont...)

Scroll Bar (Cont…):

- It defines the following constructors:

  1. Scrollbar( ) throws HeadlessException //Create a vertical scroll bar
  2. Scrollbar(int *style*) throws HeadlessException
  3. Scrollbar(int *style*, int *initialValue*, int *thumbSize*, int *min*, int *max*) throws HeadlessException

  In the second and third forms, if *style* is Scrollbar.VERTICAL, a vertical scroll bar is created. If *style* is Scrollbar.HORIZONTAL, the scroll bar is horizontal.

  In the third form, the initial value is passed in *initialValue*. The height of the thumb is passed in *thumbSize*. The minimum and maximum values of the scroll bar are specified by *min* and *max*.

- If we use one of the first two constructors, we need to set its parameters by using setValues( ) before it is used:

  void setValues(int *initialValue*, int *thumbSize*, int *min*, int *max*)

Control Fundamentals (Cont...)

Scroll Bar (Cont…):

- To obtain the current value of the scroll bar, we call getValue( ). It returns the current setting. To set the current value, call setValue( ).

```
1. int getValue( )
2. void setValue(int newValue)      //newValue is the new value for the scroll bar
```

- We can also retrieve the minimum and maximum values by getMinimum( ) and getMaximum( ) given below:

```
1. int getMinimum( )
2. int getMaximum( )
```

- By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line. We can change this increment by calling setUnitIncrement( ).
- By default, page-up and page-down increments are 10. We can change this value by calling setBlockIncrement( ).

```
1. void setUnitIncrement(int newIncr)
2. void setBlockIncrement(int newIncr)
```

Control Fundamentals (Cont...)

Scroll Bar (Cont…):

- To process scroll bar events, we need to implement the AdjustmentListener interface.
- Each time a user interacts with a scroll bar, an AdjustmentEvent object is generated. Its getAdjustmentType( ) method can be used to determine the type of the adjustment.
- The types of adjustment events are as follows:

Adjustment Events	Description
BLOCK_DECREMENT	A page-down event has been generated.
BLOCK_INCREMENT	A page-up event has been generated.
TRACK	An absolute tracking event has been generated.
UNIT_DECREMENT	The line-down button in a scroll bar has been pressed.
UNIT_INCREMENT	The line-up button in a scroll bar has been pressed.

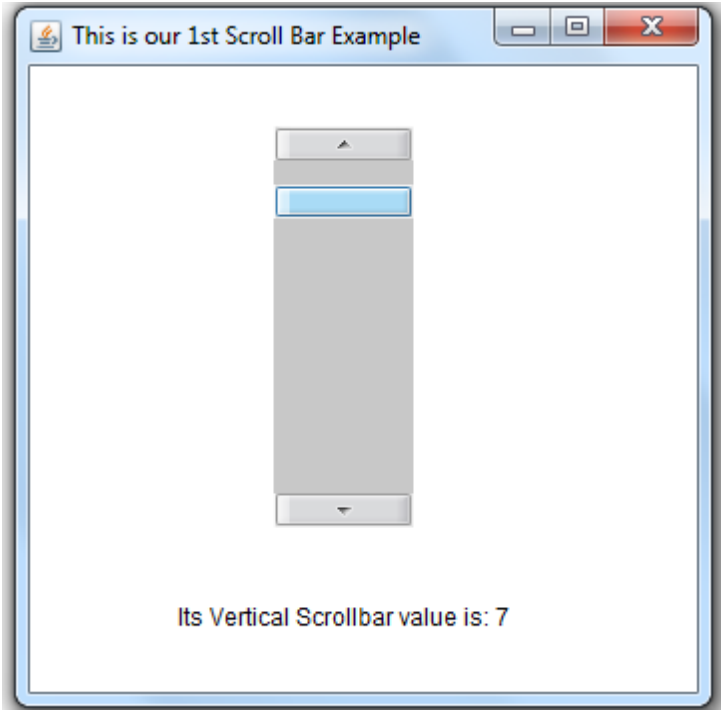
Control Fundamentals (Cont...)

- **Scroll Bar (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class Scrollbar1
{
  Scrollbar1()
  {
    Frame f=new Frame();
    f.setTitle("This is our 1st Scroll Bar Example");
    final Label l= new Label();
    l.setAlignment(Label.LEFT);
    l.setBounds(80, 280, 550, 50);
    Button b=new Button("Click Here");
    final Scrollbar sr=new Scrollbar();
    sr.setBounds(130, 60, 70, 200);
    f.add(l);
    f.add(sr);
    f.setSize(350, 350);
    f.setLayout(null);
    f.setVisible(true);
    f.addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        f.dispose();
      }
    }
    );
    sr.addAdjustmentListener(new AdjustmentListener()
    {
      public void adjustmentValueChanged(AdjustmentEvent e)
      {
        l.setText("Its Vertical Scrollbar value is: " + sr.getValue());
      }
    });
  }
  public static void main(String args[])
  {
    new Scrollbar1();
  }
}
```

Control Fundamentals (Cont...)

- **Scroll Bar (Cont…):**
  - **Output**



  - **If we scroll it, Vertical Scrollbar value is changed.**

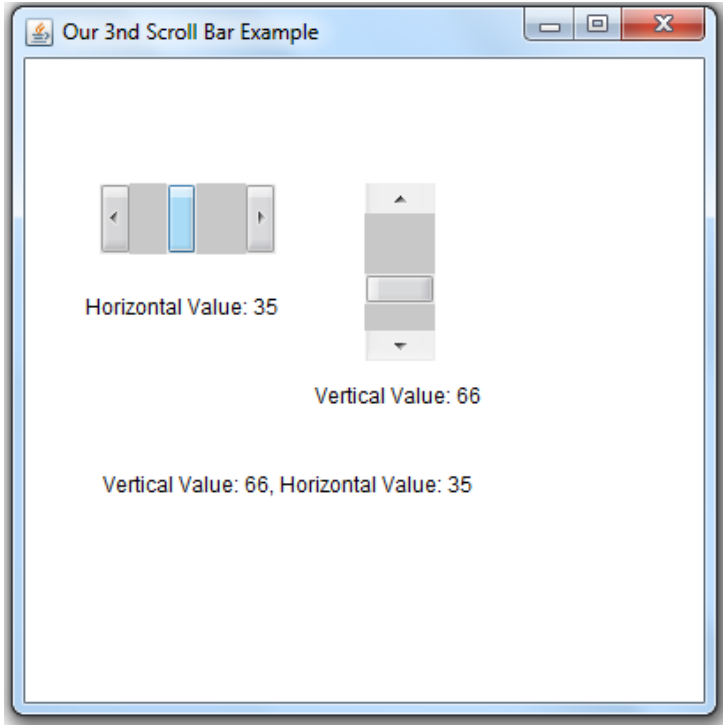
Control Fundamentals (Cont...)

Scroll Bar (Cont…):

```
import java.awt.*;
import java.awt.event.*;
public class Scrollbar3
{
  public Scrollbar3()
  {
    Frame f=new Frame();
    f.setTitle("Our 3nd Scroll Bar Example");
    final Scrollbar sr1=new Scrollbar(Scrollbar.VERTICAL, 60, 5, 0, 100);
    sr1.setBounds(200, 100, 40, 100);
    final Scrollbar sr2=new Scrollbar(Scrollbar.HORIZONTAL, 40, 20, 0, 100);
    sr2.setBounds(50, 100, 100, 40);
    Label l1=new Label();
    l1.setBounds(170, 200, 100, 40);
    Label l2=new Label();
    l2.setBounds(40, 150, 200, 40);
    Label l3=new Label();
    l3.setBounds(50, 250, 250, 40);
    f.add(sr1); f.add(sr2); f.add(l1); f.add(l2); f.add(l3);
    f.setSize(400, 400);  f.setLayout(null);  f.setVisible(true);
    f.addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        f.dispose();
      }
    } );

    sr1.addAdjustmentListener(new AdjustmentListener()
    {
      public void adjustmentValueChanged(AdjustmentEvent e)
      {
        l1.setText("Vertical Value: " + sr1.getValue());
        l3.setText("Vertical Value: " + sr1.getValue() + ", Horizontal Value: " + sr2.getValue());
      }
    });

    sr2.addAdjustmentListener(new AdjustmentListener()
    {
      public void adjustmentValueChanged(AdjustmentEvent e)
      {
        l2.setText("Horizontal Value: " + sr2.getValue());
        l3.setText("Vertical Value: " + sr1.getValue() + ", Horizontal Value: " + sr2.getValue());
      }
    });
  }
  public static void main(String args[])
  {
    new Scrollbar3();
  }
}
```

Control Fundamentals (Cont...)

- **Scroll Bar (Cont…):**
  - **Output**



- **If we change any scrollbar value, it will be shown in both places.**

Layout Manager

- We can manually give a layout to the output window.
- Each Container object has a layout manager linked with it.
- Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.
- A layout manager is an instance of any class that implements the LayoutManager interface. It is set by the setLayout( ) method.

  void setLayout(LayoutManager *layoutObj*)

  Here, *layoutObj* is a reference to the desired layout manager
- Each layout manager keeps track of a list of components that are stored by their names.

Layout Manager (Cont…)

- Each time the layout manager is notified, when we add a component to a container.
- Whenever a container needs to be resized, the layout manager is consulted via its minimumLayoutSize( ) and preferredLayoutSize( ).
- Each component that is being managed by a layout manager contains the getPreferredSize( ) and getMinimumSize( ).
- Java supports several predefined layouts by using LayoutManager classes, such as BorderLayout, FlowLayout, GridLayout, CardLayout, etc.

Layout Manager (Cont…)

- **FlowLayout:**
  - FlowLayout is the default layout manager.
  - FlowLayout implements a simple layout style, which is similar to how words flow in a text editor.
  - The direction of the layout is governed by the container's component orientation property. By default left to right, top to bottom.
  - A small space is left between each component, above and below, as well as left and right.

Layout Manager (Cont…)

FlowLayout (Cont…):

- There are three constructors for FlowLayout:

1. FlowLayout( ) //It creates the default layout, which centers components and leaves five pixels of space between each component.
2. FlowLayout(int *how*) //This specifies how each line is specified by *how*.

   The valid values of how are:

   FlowLayout.LEFT
   FlowLayout.CENTER
   FlowLayout.RIGHT
   FlowLayout.LEADING
   FlowLayout.TRAILING
3. FlowLayout(int *how*, int *horz*, int *vert)* //Specifies the horizontal and vertical space left between components in *horz* and *vert*, respectively.

Layout Manager (Cont…)

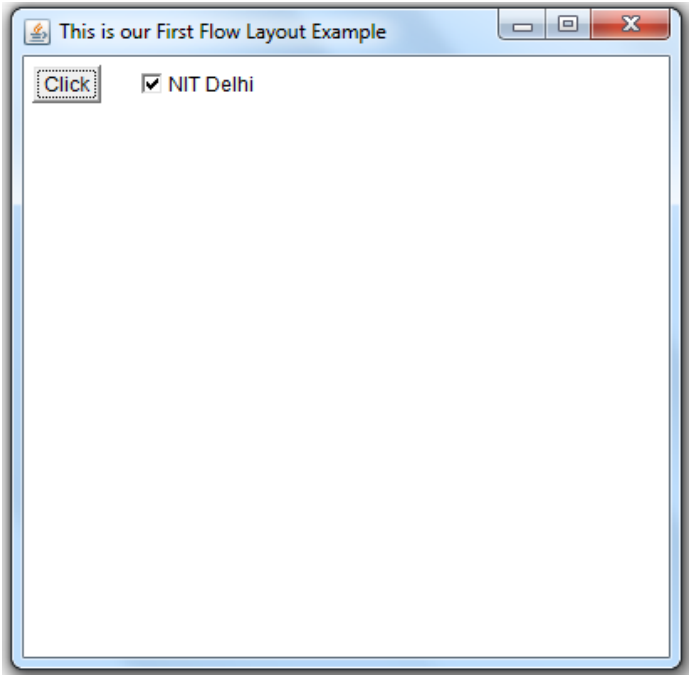
- **FlowLayout (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class Flowlayout1
{
  Frame f;
  public Flowlayout1()
  {
    f=new Frame ("This is our First Flow Layout Example");
    Button b = new Button("Click");
    Checkbox c = new Checkbox("NIT Delhi", true);
    Label l = new Label();
    f.add(b);
    f.add(l);
    f.add(c);

    b.addActionListener(new ActionListener()
    {
      public void actionPerformed(ActionEvent e)
      {
        f.setBackground(Color.GRAY);
        l.setText("Have Fun...");
        l.setBounds(40, 100, 100, 30);
      }
    } );

    f.setSize(400, 400);
    f.setLayout(new FlowLayout(FlowLayout.LEFT));
    f.setVisible(true);

    f.addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        f.dispose();
      }
    } );
  }
  public static void main(String[] args)
  {
    new Flowlayout1();
  }
}
```

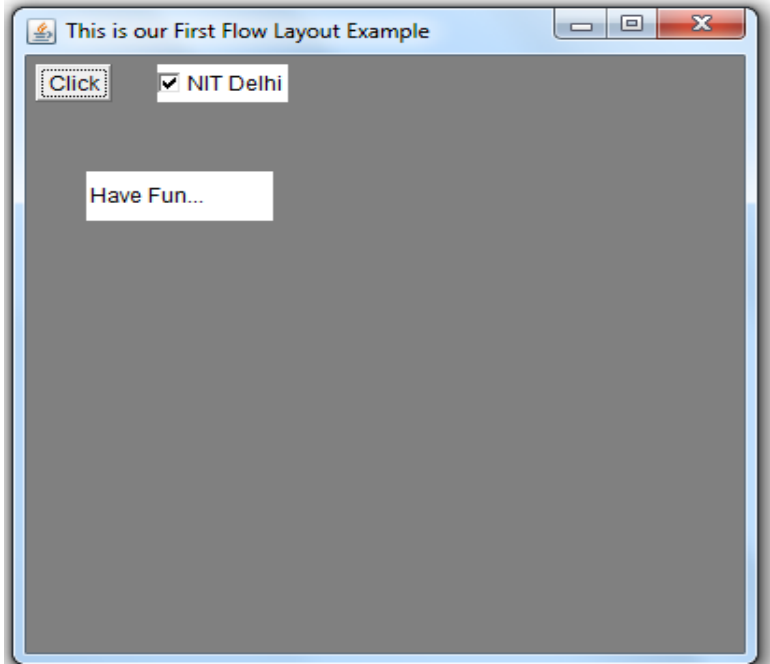
Layout Manager (Cont…)

- **FlowLayout (Cont…):**
  - **Output**

Before Clicking



After Clicking



Layout Manager (Cont…)

BorderLayout:

- This class implements a common layout style for top-level windows.
- It has four narrow, fixed-width components at the edges and one large area in the center.
- The four sides are referred by north, south, east and west. The middle area is called the center. The constructors defined as:

1. BorderLayout( ) //Create a default border layout
2. BorderLayout(int *horz*, int *vert*) //Allows to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

Layout Manager (Cont…)

BorderLayout (Cont…):

- BorderLayout defines the five constants that specify the regions:
  1. BorderLayout.CENTER
  2. BorderLayout.SOUTH
  3. BorderLayout.EAST
  4. BorderLayout.WEST
  5. BorderLayout.NORTH.
- When adding components, we use the above mentioned constants with the following form of add( ) that is defined by Container:

  void add(Component *compObj*, Object *region*)

  Here, *compObj* is the component to be added, and *region* specifies where the component will be added.

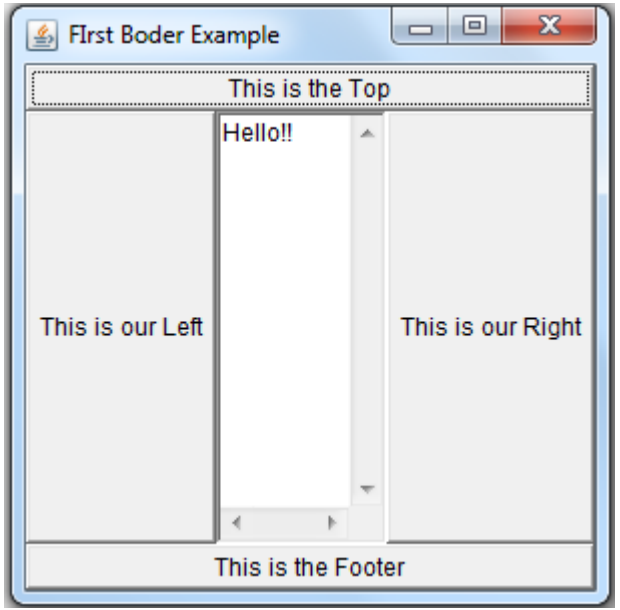
Layout Manager (Cont…)

- **BorderLayout (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class Border1 extends Frame
{
  public Border1()
  {
    setTitle("FIrst Boder Example");
    setSize(300, 300);
    setLayout(new BorderLayout());
    setVisible(true);
    Button b1 = new Button("This is the Top");
    add(b1, BorderLayout.NORTH);
    Button b2= new Button("This is the Footer");
    add(b2, BorderLayout.SOUTH);
    Button b3= new Button("This is our Right");
    add(b3, BorderLayout.EAST);
    Button b4= new Button("This is our Left");
    add(b4, BorderLayout.WEST);
    TextArea t= new TextArea("Hello!!");
    add(t, BorderLayout.CENTER);

    addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        dispose();
      }
    }
    );
  }
  public static void main(String args[])
  {
    new Border1();
  }
}
```

Layout Manager (Cont…)

- **BorderLayout (Cont…):**
  - **Output**



Layout Manager (Cont…)

GridLayout:

- In GridLayout, components are align in two-dimensional grid.
- When we instantiate a GridLayout, we can define the number of rows and columns. The constructors of GridLayout are given below:

1. GridLayout( ) //Creates a single column grid layout
2. GridLayout(int *numRows*, int *numColumns*) //Creates a grid layout with specific number of rows and column
3. GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*) //Creates a grid layout with specified horizontal and vertical space left between components in *horz* and *vert*, respectively. Here, either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimited-length columns and vice versa.

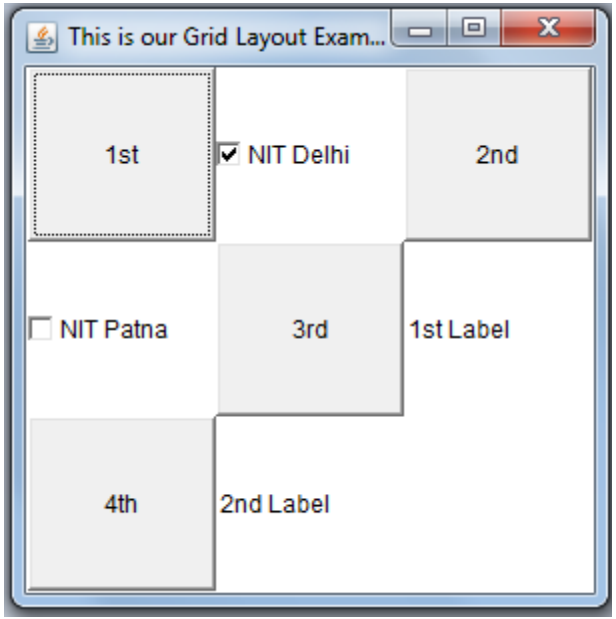
Layout Manager (Cont…)

- **GridLayout (Cont…):**

```
import java.awt.*;
import java.awt.event.*;
public class Gridlayout1
{
  Frame f;
  public Gridlayout1()
  {
    f=new Frame("This is our Grid Layout Example");

    Button b1=new Button("1st");
    Checkbox c1=new Checkbox("NIT Delhi", true);
    Button b2=new Button("2nd");
    Checkbox c2=new Checkbox("NIT Patna");
    Button b3=new Button("3rd");
    Label l1= new Label("1st Label");
    Button b4=new Button("4th");
    Label l2= new Label("2nd Label");

    f.add(b1);f.add(c1);f.add(b2);f.add(c2);
    f.add(b3);f.add(l1);f.add(b4);f.add(l2);

    f.setSize(300,300);
    f.setLayout(new GridLayout(3,3));
    f.setVisible(true);

    f.addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        f.dispose();
      }
    } );
  }
  public static void main(String[] args)
  {
    new Gridlayout1();
  }
}
```

Layout Manager (Cont…)

- **GridLayout (Cont…):**
  - **Output**




thank
you . . .

Slides are prepared from various sources, such as Book, Internet Links and many more.