UNIT -5. Graphics Programming:

What is Swing?

- Swing is a set of program component s for Java programmers that provide the ability to create graphical user interface (GUI) components, such as buttons and scroll bars, that are independent of the windowing system for specific operating system . Swing components are used with the Java Foundation Classes (JFC).
- The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.
- Java Swing is a lightweight Graphical User Interface (GUI) toolkit that includes a rich set of widgets. It includes package lets you make GUI components for your Java applications, and It is platform independent.
- The Swing library is built on top of the Java Abstract Window Toolkit (**AWT**), an older, platform dependent GUI toolkit. You can use the Java GUI components like button, textbox, etc. from the library and do not have to create the components from scratch.

- Java offers two standard libraries for graphical user interface (GUI).
- The **first one** is the simpler java.awt package, which contains classes to create windows (java.awt.Window), buttons (java.awt.Button), textfields (java.awt.TextField), and so on. These classes are simple wrappers to the platform's GUI objects. For instance, a window created with awt in Java would look like a Windows window in Windows, and like a Gnome window under GNU/Linux Gnome interface. The advantage is that the "look-and-feel" is exactly like the native system. The disadvantage is that since there is a big variety of platforms, this package does not allow one to finely tune the graphical components, since not every feature is allowed in all platforms.

- We will rather look at the second library which can be found in the javax.swing package. This contains classes with similar names to the awt package, but they are prefixed by the letter J. For instance, a Java swing window is the javax.swing.JWindow class, a button is the javax.swing.JButton class. The difference with the awt is that swing objects are drawn by Java, so they look and act exactly the same in all systems. This offers better control over the components.

Difference between AWT and Swing

No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

Creating Frame

- A Frame is a top-level window with a title and a border.
- The javax.swing.JFrame class is a type of container which inherits the java.awt.Frame class. JFrame works like the main window where components like labels, buttons, textfields are added to create a GUI.
- JFrame has the option to hide or close the window with the help of setDefaultCloseOperation(int) method.

Constructor	Description
JFrame()	It constructs a new frame that is initially invisible. To make the frame visible, invoke setVisible(true) on it.
JFrame(GraphicsConfiguration gc)	It creates a Frame in the specified GraphicsConfiguration of a screen device and a blank title.
JFrame(String title)	It creates a new, initially invisible Frame with the specified title.
JFrame(String title, GraphicsConfiguration gc)	It creates a JFrame with the specified title and the specified GraphicsConfiguration of a screen device.
Method	**Purpose**
void setDefaultCloseOperation(int) int getDefaultCloseOperation()	Possible choices are: DO_NOTHING_ON_CLOSE HIDE_ON_CLOSE DISPOSE_ON_CLOSE EXIT_ON_CLOSE

- DO_NOTHING_ON_CLOSE (defined in WindowConstants) :- Don't do anything; require the program to handle the operation in the windowClosing method of a registered WindowListener object.
- HIDE_ON_CLOSE (defined in WindowConstants) :- Automatically hide the frame after invoking any registered WindowListener objects.
- DISPOSE_ON_CLOSE (defined in WindowConstants): Automatically hide and dispose the frame after invoking any registered WindowListener objects.
- EXIT_ON_CLOSE (defined in JFrame): Exit the application using the System exit method. Use this only in applications.

Example of Creation of Frame

```
import javax.swing.*;
class Frame1
{
    public static void main(String args[])
    {
                    int x=120,y=220;
                    JFrame f = new JFrame();
                    f.setSize(500,500);
                    f.setTitle(“My Frame");
                    f.setLocation(x,y);
                    f.setVisible(true);
    }
}
```

Example of Creation of Frame

```
import javax.swing.*;
class Frame1
{
    public static void main(String args[])
    {
                int x=120,y=220;
                JFrame f = new JFrame();
                f.setSize(500,500);
                f.setTitle(“My Frame");
                f.setLocation(x,y);
                f.setVisible(true);
    }
}
```

Positioning a Frame

- The JFrame class itself has only a few methods for changing how frames look. Of course, through the magic of inheritance, most of the methods for working with the size and position of a frame come from the various superclasses of JFrame.
- The most important methods related to positioning a Frame are:
  1) The setLocation and setBounds methods for setting the position of the frame
  2) The setResizable method, which takes a boolean to determine if a frame will be resizeable by the user

- **void setLocation(int x, int y):-** Move the component to new location. The x and y Coordinates
- **void setSize(int width, int height):-**
- Resizes the component to the specified width and height.
- **setBounds()**
- The **setBounds()** method needs four arguments. The first two arguments are **x and y coordinates** of the **top-left corner** of the component, the third argument is the **width** of the component and the fourth argument is the **height** of the component.
- **Syntax**
- **setBounds(int x-coordinate, int y-coordinate, int width, int height)**

- You can determine the screen resolution (screen size) using the Toolkit class. This method call returns the screen resolution in pixels, and stores the results in a Dimension object, as shown here:
- **Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();**
- You can then get the screen width and height as int's by directly accessing the width and height fields of the Dimension class, like this:

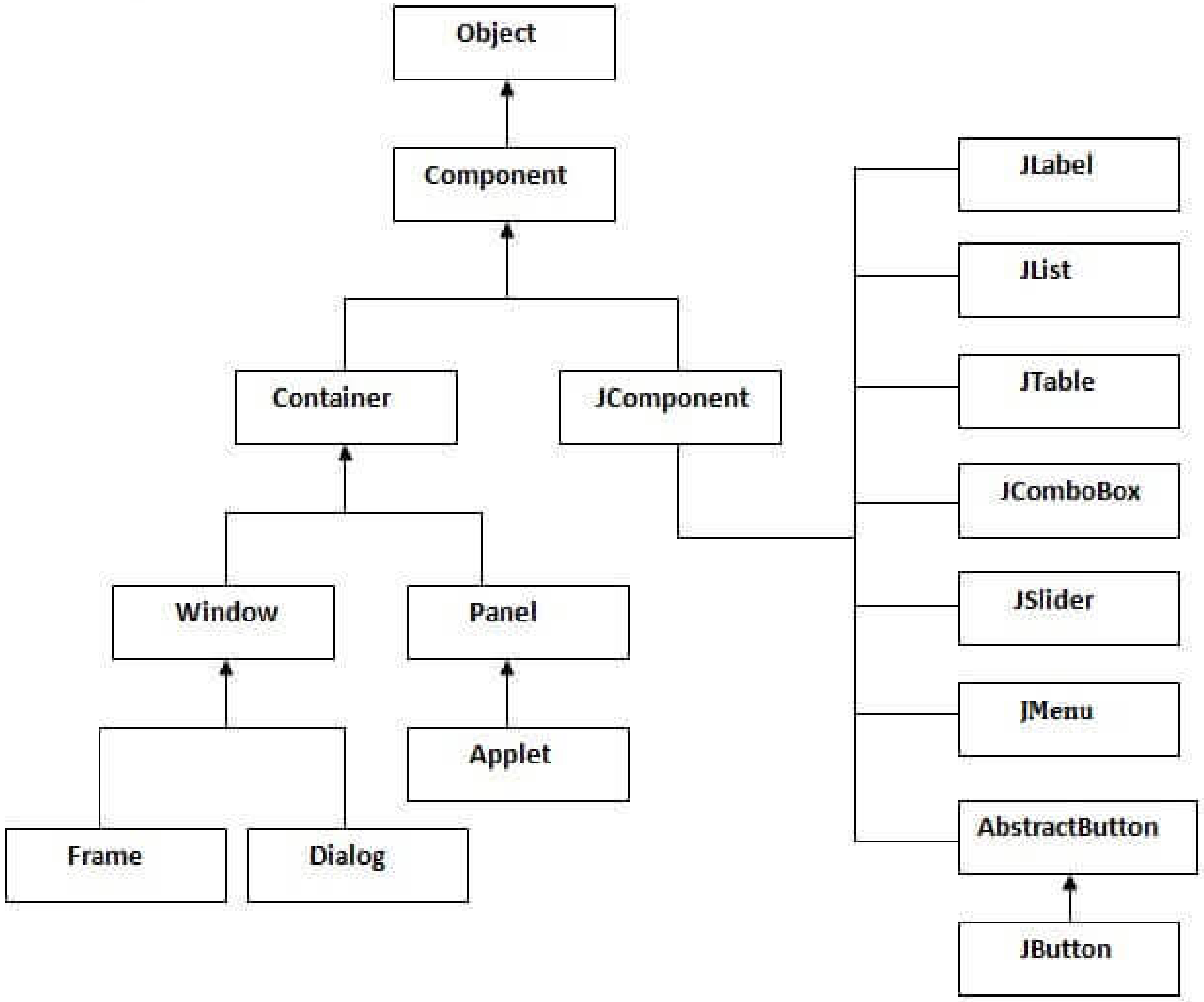
```
screenHeight = screenSize.height;
screenWidth = screenSize.width;
```

- once you have the screen dimensions, you can easily set the JFrame size to half the height and half the width of the screen, like this:
- frame.setSize(screenWidth / 2, screenHeight / 2);

- The Toolkit Class has a method called getScreenSize that returns the screen size as a Dimension object. A Dimension object simultaneously stores a width and height.

```
•	import java.awt.*;
•	import java.awt.event.*;
•	import javax.swing.*;
•	public class CenteredFrameTest
•	{
•		public static void main (String[] args)
•		{
•				CenteredFrame frame=new CenteredFrame();
•				frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE );
•				frame.show();
•		}
•	}
•	class CenteredFrame extends JFrame
•	{
•		public  CenteredFrame()
•		{
•		//get screen dimensions
•				Toolkit kit=Toolkit.getDefaultToolkit();
•				Dimension screenSize=kit.getScreenSize();
•				int screenHieght=screenSize.height;
•				int screenWidth=screenSize.width;
•		// Center frame in screen
•				setSize(screenWidth/2, screenHieght/2);
•				setLocation(screenWidth/4, screenHieght/4);
•		}
•	}
```

Hierarchy of Java Swing classes

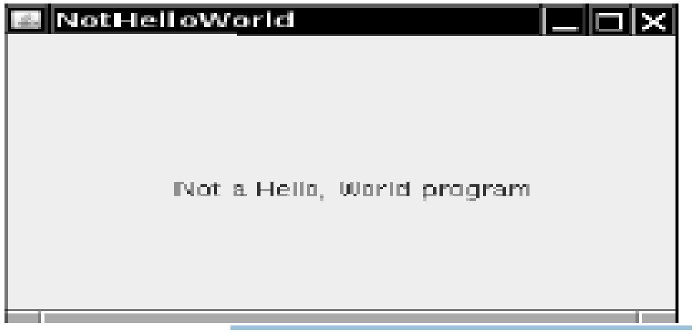


Commonly used Methods of Component class

Method	Description
public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

Displaying a Information in Panel

- **A frame that displays information**



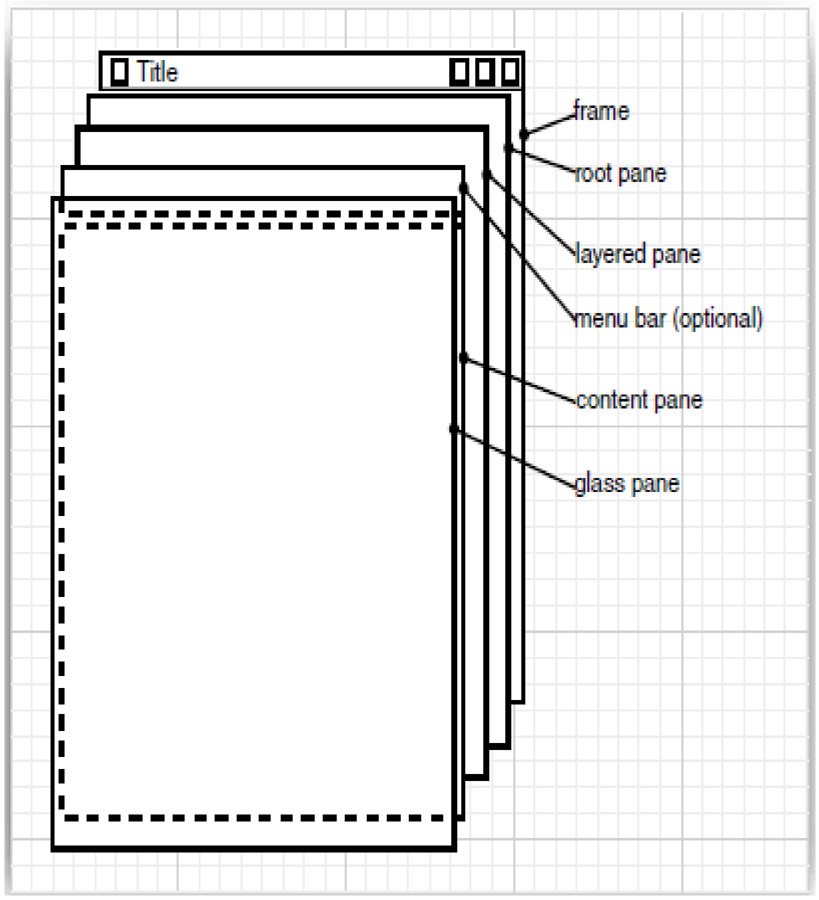
Not a Hello, World program

- ➢ You could draw the message string directly onto a frame, but that is not considered good programming practice.
- ➢ In Java, frames are really designed to be containers for components such as a menu bar and other user interface elements.
- ➢ You normally draw on another component which you add to the frame.

- The structure of a JFrameis surprisingly complex. Look at Figure below, which shows the makeup of a JFrame.
- As you can see, four panes are layered in a JFrame. The root pane, layered pane, and glass pane are of no interest to us; they are required to organize the menu bar and content pane and to implement the look and feel.
- The part that most concerns Swing programmers is the *content pane*. When designing a frame, you add components into the content pane, using code such as the following:

```
Container contentPane =  frame.getContentPane();
Component c = . . .;
 contentPane.add(c);
```

- **Internal structure of a JFrame**



- We want to add a single panel to the content pane onto which we will draw our message.
- Panel are implemented by the JPanel class.
- To Draw on a panel, you will need to

I) Define a class the extends JPanel;

II) Override the paintComponent method in that class

```
Class MyPanel extends JPanel
{
        public void paintComponent (Graphics g)
        {
                …//code for drawing will go here
        }
}
```

- “Do not use JFrame.add(). Use JFrame.getContentPane().add()
- you can simply use the call
- frame.add(c);
- we want to add a single component to the frame onto which we will draw our message. To draw on a component, you define a class that extends JComponent and override the paintComponentmethod in that class.

- The paint Component method takes one parameter of type Graphics.
- A Graphics object remembers a collection of settings for drawing images and text, such as the font you set or the current color. All drawing in Java must go through a Graphics object. It has methods that draw patterns, images, and text.
- Here’s how to make a component onto which you can draw:

```
class MyComponent extends JComponent
 {
          public void  paintComponent(Graphics g)
         {         code for drawing
          }
 }
```

- Each time a window needs to be redrawn, no matter what the reason, the event handler notifies the component. This causes the paintComponent methods of all components to be executed.
- Never call the paintComponentmethod yourself. It is called automatically whenever a part of your application needs to be redrawn, and you should not interfere with this automatic process.
- Displaying text is considered a special kind of drawing. The Graphics class has a drawstring method that has the following syntax:
- g.drawString(text, x, y)

- paintComponentmethod Example

```
classNotHelloWorldComponent  extends JComponent
{
   public void paintComponent(Graphics g)
   {
g.drawString("Not a Hello, World program", MESSAGE_X, 
MESSAGE_Y);
   }
public static final int  MESSAGE_X = 75;
public static final int  MESSAGE_Y = 100;
}
```

- ***javax.swing.Jframe***
- Container getContentPane()
  returns the content pane object for this JFrame.
- Component add(Component c)
  adds and returns the given component to the content pane of this frame.
- void repaint()
  causes a repaint of the component “as soon as possible.”
- ***javax.swing.Jcomponent***
- public void repaint(int x, int y, int width, int height)
  causes a repaint of a part of the component “as soon as possible.”
- void paintComponent(Graphics g)
  overrides this method to describe how your component needs to be painted.

Working with 2D Shapes

- Starting with Java 1.0, the Graphics class had methods to draw lines, rectangles, ellipses, and so on.
- But those drawing operations are very limited.
- For example, you cannot vary the line thickness and you cannot rotate the shapes.
- Java SE 1.2 introduced the *Java 2D* library, which implements a powerful set of graphical operations.

- To draw shapes in the Java 2D library, you need to obtain an object of the Graphics2D class.
- This class is a subclass of the Graphics class.
- Ever since Java SE 2, methods suchas paintComponent automatically receive an object of the Graphics2D class.
- Simply use acast, as follows:

```
public void  paintComponent(Graphics g)
 {
          Graphics2D g2 =  (Graphics2D) g;
            . . .
}
```

- The Java 2D library organizes geometric shapes in an object-oriented fashion. In particular, there are classes to represent lines, rectangles, and ellipses:

1) Line2D

2) Rectangle2D

3) Ellipse2D

```
Rectangle2D rect = . . .;
 g2.draw(rect);
```

- Using the Java 2D shape classes introduces some complexity. Unlike the 1.0 draw methods, which used integer pixel coordinates, the Java 2D shapes use floating point coordinates.
- However, manipulating float values is some times inconvenient for the programmer because the Java programming language is adamant about requiring casts when converting double values into float values.
- For example, consider the following statement:
- float f = 1.2; // Error
- This statement does not compile because the constant 1.2 has type double, and the compiler is nervous about loss of precision. The remedy is to add an **F** suffix to the floating point constant :
- float f = 1.2F; // Ok

- Now consider this statement:

Rectangle2D r = . . .

float f = r.getWidth(); // Error

- This statement does not compile either, for the same reason. The getWidthmethod returns a double. This time, the remedy is to provide a cast:

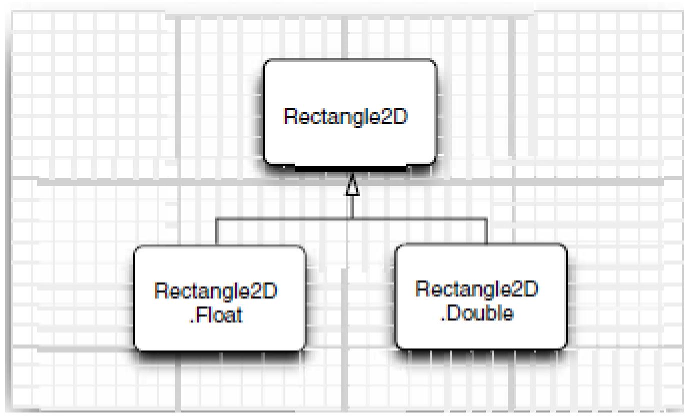
float f = (float) r.getWidth(); // Ok

- The designers of the 2D library decided to supply *two versions* of each shape class: one with float coordinates, and one with double coordinates.
- the Rectangle2D class. This is an abstract class with two concrete subclasses,

1) Rectangle2D.Float

2) Rectangle2D.Double

• 2D rectangle classes



- When you construct a Rectangle2D.Float object, you supply the coordinates as float numbers. For a Rectangle 2D .Double object, you supply them as double numbers.

Rectangle2D.Float floatRect = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);
Rectangle2D.Double doubleRect = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);

- Further more, there is a Point2D class with sub classes Point2D .Float and

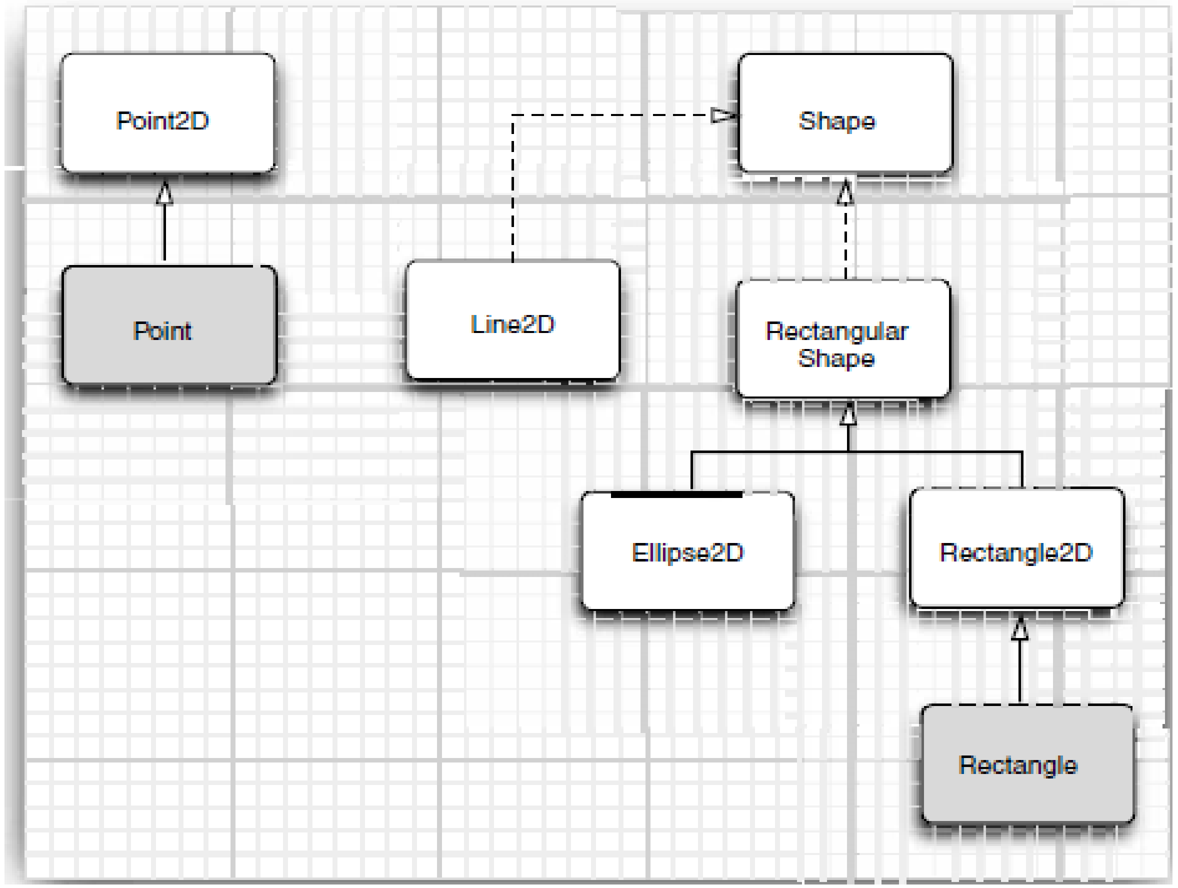
Point2D .Double.

- Here is how to make a point object.

Point2D p = new Point2D.Double(10, 20);

- The Point2D class is very useful—it is more object oriented to work with Point2D objects than with separate *x*- and *y*- values. Many constructors and methods accept Point2D parameters.

- **Relationships between the shape classes**



- **java.awt.geom.RectangularShape**
- doublegetCenterX()
- doublegetCenterY()
- doublegetMinX()
- doublegetMinY()
- doublegetMaxX()
- doublegetMaxY()
- Returns the center, minimum, or maximum *x*- or *y*-value of the enclosing rectangle.
- doublegetWidth()
- doublegetHeight()
  returns the width or height of the enclosing rectangle.
- doublegetX() double getY()
- Returns the *x*- or *y*-coordinate of the top-left corner of the enclosing rectangle.

- ***java.awt.geom.Rectangle2D.Double***
- Rectangle2D.Double(double x, double y, double w, double h)
- constructs a rectangle with the given top-left corner, width, and height.
- Rectangle2D.Float(float x, float y, float w, float h) constructs a rectangle with the given top-left corner, width, and height
- ***java.awt.geom.Ellipse2D.Double***
- Ellipse2D.Double(double x, double y, double w, double h)
- constructs an ellipse whose bounding rectangle has the given top-left corner, width, and height.

- ***java.awt.geom.Point2D.Double***
- Point2D.Double(double x, double y)
- constructs a point with the given coordinates.
- Line2D.Double(Point2D start, Point2D end)
- Line2D.Double(double startX, double startY, double endX, double endY). constructs a line with the given start and end points

Color

- The set Paint method of the Graphics2D class lets you select a color that is used for all subsequent drawing operations on the graphics context. For example:
- g2.setPaint(Color.RED);
- g2.drawString("Warning!", 100, 100);
- You can fill the interiors of closed shapes (such as rectangles or ellipses) with a color.
- Rectangle2D rect = . . .;
- g2.setPaint(Color.RED);
- g2.fill(rect); // fills rect with red color
- To draw in multiple colors, you select a color, draw or fill, then select another color, and draw or fill again.
- You define colors with the **Color** class.
- The java .awt .Colorclass offers predefined constants for the following 13 standard colors:
- BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE, YELLOW

- You can specify a custom color by creating a Color object by its red, green, and blue components. Using a scale of 0–255 (that is, one byte) for the redness, blueness, and greenness, call the Color constructor like this:

  Color(int redness, int greenness, int blueness)

  Here is an example of setting a custom color:

  g2.setPaint(new Color(0, 128, 128)); // a dull blue-green

  g2.drawString("Welcome!", 75, 125);
- To set the *background color,* you use the set Background method of the Component class, an ancestor of JComponent.
- MyComponent p = new MyComponent();
- p.setBackground(Color.PINK);

- ***java.awt.Color***
- Color(int r, int g, int b)
  creates a color object.
- Color getColor()
- void setColor(Color c)
  gets or sets the current color. All subsequent graphics operations will use the new color.
- Parameters: r The red value (0–255), gThe green value (0–255), bThe blue value (0–255)
- ***java.awt.Graphics2D***
- Paint getPaint()
- void setPaint(Paint p)
  gets or sets the paint property of this graphics context. The Color class implements the Paint interface. Therefore, you can use this method to set the paint attribute to a solid color.
- void fill(Shape s)
  fills the shape with the current paint.
- ***java.awt.Component1.0***
- Color getBackground()
- void setBackground(Color c)
- gets or sets the background color.<>
- Color getForeground()
- void setForeground(Color c)

Special Fonts for Text

- you want to show text in a different font. You specify a font by its *font face name*. A font face name is composed of a *font family name.*
- To find out which fonts are available on a particular computer, call the
- **getAvailableFontFamilyNames** method of the **GraphicsEnvironment** class.
- The method returns an **array** of strings that contains the names of all available fonts.
- o obtain an instance of the
- Graphics Environment class that describes the graphics environment of the user’s system, use the static getLocalGraphicsEnvironmentmethod.

- Thus, the following program prints the names of all fonts on your system:
- importjava.awt.*;
- public class ListFonts
- {
- public static void main(String[] args)
- {
- String[] fontNames = GraphicsEnvironment
- .getLocalGraphicsEnvironment()
- .getAvailableFontFamilyNames();
- for (String fontName : fontNames)
- System.out.println(fontName);
- }
- }

- To draw characters in a font, you must first create an object of the class Font. You specify the font face name, the font style, and the size.
- Font(String name, int style, int size);
- Here is an example of how you construct a Font object:
- Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
- You specify the style (plain, **bold**, *italic,* or ***bold italic***) by setting the second Font constructor argument to one of the following values:
- Font.PLAIN
- Font.BOLD
- Font.ITALIC
- Font.BOLD + Font.ITALIC

- Classes and Methods
- Font(String name, int style, int size)
  creates a new font object.
- String getFontName()
  gets the font face name (such as “Helvetica Bold”).
- String getFamily()
- String getName()
  gets the logical name (such as “SansSerif”) if the font was created with a logical font name; otherwise, gets the font face name.
- Font getFont()
- void setFont(Font font)
  gets or sets the current font. That font will be used for subsequent text-drawing operations.
- Void drawString(String str, int x, int y)

UNIT -5. Event Handling and User Interface Components with Swing:

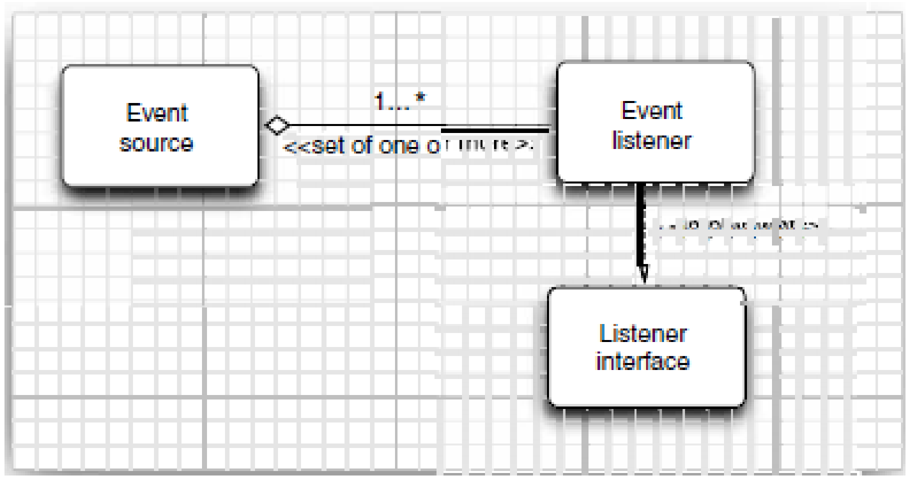
Basics of Event Handling

- Any operating environment that supports GUIs constantly monitors events such as key strokes or mouse clicks.
- The operating environment reports these events to the programs that are running.
- Each program then decides what, if anything, to do in response to these events.
- In languages like Visual Basic, the correspondence between events and code is obvious. One writes code for each specific event of interest and places the code in what is usually called an ***event procedure.***
- Basic button named “HelpButton” would have a Help Button _Click event procedure associated with it. The code in this procedure executes whenever that **button is clicked**.
- **Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. The java.awt.event package provides many event classes and Listener interfaces for event handling.**

- The Java programming environment takes an approach somewhat between the VisualBasic approach and the raw C approach in terms of power and, therefore, in resulting complexity. Within the limits of the events that the AWT knows about, you completely control how events are transmitted from the *event sources* (such as buttons or scrollbars)to *event listeners*.
- Event sources have methods that allow you to register event listeners with them.
- When an event happens to the source, the source sends a notification of that event to all the listener objects that were registered for that event.
- Different event sources can produce different kinds of events. For example, a button can send Action Event objects, whereas a window can send Window Event objects.

- To sum up, here’s an overview of how event handling in the AWT works:
- A listener object is an instance of a class that implements a special interface called(naturally enough) a *listener interface.*
- **An event source is an object that can register listener objects and send them event objects.**
- **The event source sends out event objects to all registered listeners when that event occurs.**
- **The listener objects will then use the information in the event object to determine their reaction to the event.**

• Relationship between event sources and listeners



Here is an example for specifying a listener:

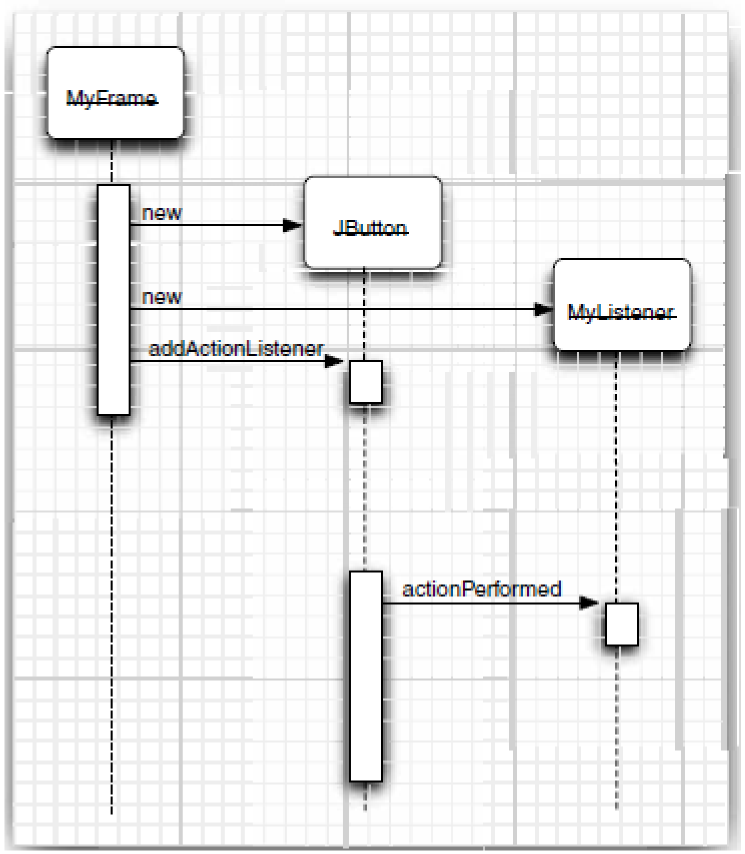
```
ActionListener listener =  . . .;
JButton button = new  JButton("Ok");
button.addActionListener(listener);
```

Now the listener object is notified whenever an “action event” occurs in the button. For buttons, as you might expect, an action event is a button click.

- To implement the Action Listener interface, the listener class must have a method called actionPerformed that receives an ActionEvent object as a parameter.
- class MyListener implements Action Listener
- {
- . . .public void actionPerformed(ActionEvent event)
- {
- // reaction to button click goes here
- . . .
- }
- }

- Whenever the user clicks the button, the JButton object creates an Action Event object and calls listener.action Performed (event), passing that event object.
- An event source such as abutton can have multiple listeners. In that case, the button calls the action Performed methodsof all listeners whenever the user clicks the button.
- the interaction between the event source, event listener, and event object.

Event notification



Example: Handling a Button Click

- As a way of getting comfortable with the event delegation model, let’s work through all details needed for the simple example of responding to a button click. For this example, we will show a panel populated with three buttons. Three listener objects are added as action listeners to the buttons.
- With this scenario, each time a user clicks on any of the buttons on the panel, the associated listener object then receives an ActionEvent that indicates a button click.
- You create a button by specifying a label string, an icon, or both in the button constructor.
- JButton yellowButton = new JButton("Yellow");
- JButton blueButton = new JButton(new ImageIcon("blue-ball.gif"));

- Class ButtonPanel extends JPanel
- {

```
   public ButtonPanel()
   {
       JButton yellowButton=new JButton("Yellow");
       JButton blueButton=new JButton("Blue");
       JButton redButton=new JButton("Red");
       add(yellowButton);
        add(blueButton);
       add(redButton);
 }
 }
```

This requires classes that implement the ActionListener interface, which, as we just mentioned, has one method: action Performed, whose signature looks like this:

public void actionPerformed(ActionEvent event)

- They make the container of the event sources implement the Action Listener interface. Then, the container sets *itself* as the listener, like this:
- yellowButton.addActionListener(this);
- blueButton.addActionListener(this);
- redButton.addActionListener(this);
- Now the three buttons no longer have individual listeners. They share a single listener object, namely, the button frame. Therefore, the actionPerformed method must figure out which button was clicked.

- class ButtonFrame extends JFrame implements ActionListener
- {
- . . .public void actionPerformed(ActionEvent event)
- {
- Object source = event.getSource();
- if (source == yellowButton) . . .
- else if (source == blueButton) . . .
- else if (source == redButton) . . .
- else . . .
- }
- }
- **java.util.EventObject**
- Object getSource()
  returns a reference to the object where the event occurred.

Adapter Classes

- Not all events are as simple to handle as button clicks. In a non-toy program, you will want to monitor when the user tries to close the main frame because you don’t want your users to lose unsaved work. When the user closes the frame, you want to put up a dialog and exit the program only when the user agrees.
- When the program user tries to close a frame window, the JFrame object is the source of a Window Event. If you want to catch that event, you must have an appropriate listener object and add it to the frame’s list of window listeners.
- WindowListener listener = . . .;
-   frame.addWindowListener(listener);

- he window listener must be an object of a class that implements the Window Listener interface. There are actually **seven** methods in the WindowListener interface. The framecalls them as the responses to seven distinct events that could happen to a window. Thenames are self-explanatory, except that “iconified” is usually called “minimized” under Windows. Here is the complete Window Listener interface: public interface Window Listener

- void windowOpened(WindowEvent e);
- void windowClosing(WindowEvent e);
- void windowClosed(WindowEvent e);
- void windowIconified(WindowEvent e);
- void windowDeiconified(WindowEvent e);
- void windowActivated(WindowEvent e);
- void windowDeactivated(WindowEvent e);

- As is always the case in Java, any class that implements an interface must implement all its methods; in this case, that means implementing *seven* methods. Recall that we are only interested in one of these seven methods, namely, the windowClosing method.
- Of course, we can define a class that implements the interface, add a call to System.exit(0) in the window Closing method, and write do -nothing functions for the other six methods:

```
•  class Terminator implements WindowListener
•   {
•   public void windowClosing(WindowEvent e)
•   {
•   if (user agrees)
•   System.exit(0);
•   }
•   public void windowOpened(WindowEvent e) {}
•   public void windowClosed(WindowEvent e) {}
•   public void windowIconified(WindowEvent e) {}
•   public void windowDeiconified(WindowEvent e) {}
•   public void windowActivated(WindowEvent  e) {}
•   public void windowDeactivated(WindowEvent e) {}
•   }
```

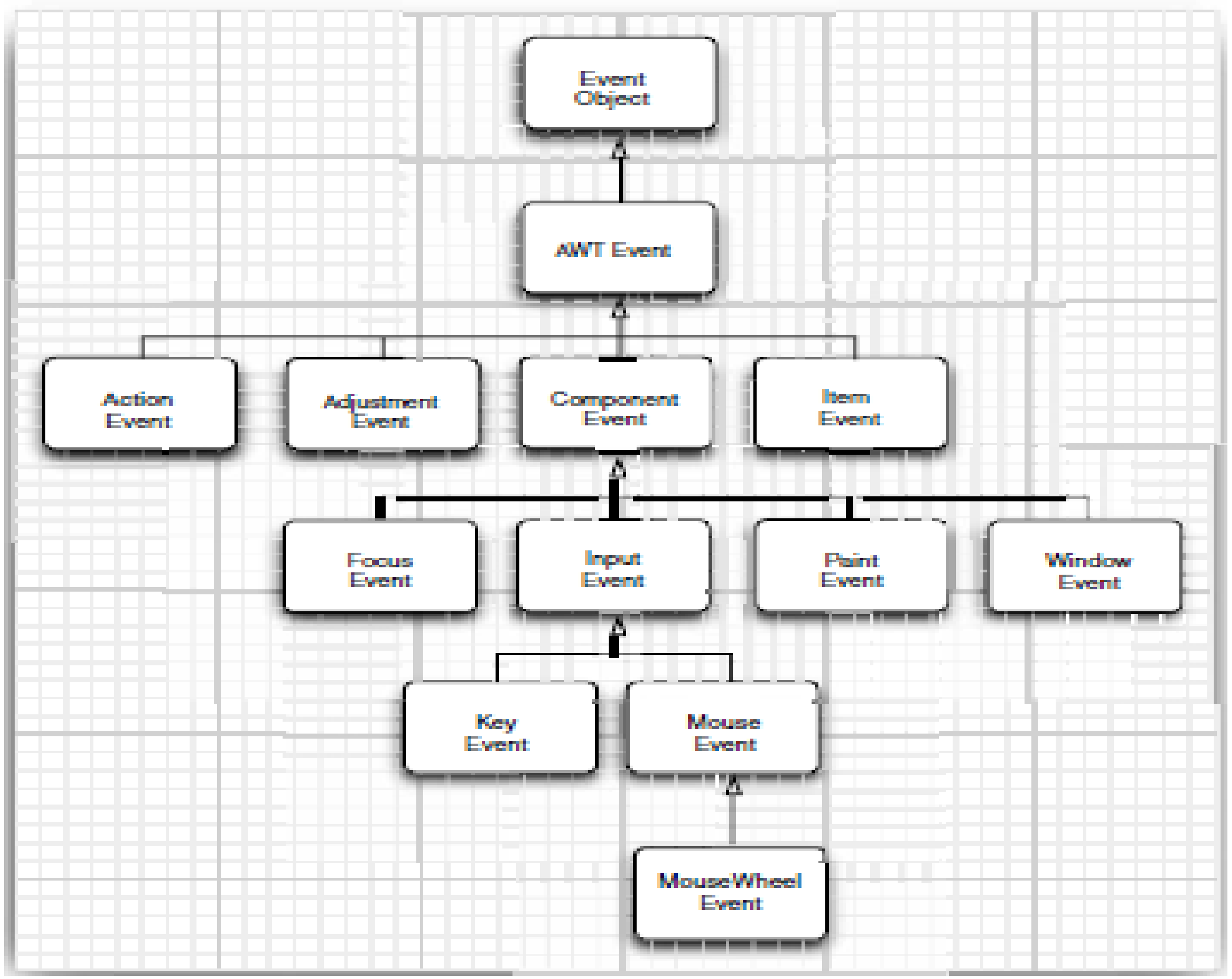
- Typing code for six methods that don’t do anything is the kind of tedious busywork that nobody likes. To simplify this task, each of the AWT listener interfaces that has more than one method comes with a companion *adapter* class that implements all the methods in the interface but does nothing with them. For example, the WindowAdapter class has seven do-nothing methods. This means the adapter class automatically satisfies the technical requirements that Java imposes for implementing the associated listener interface.

- We can extend the Window Adapter class, inherit six of the do-nothing methods, and override the window Closing method:
- class Terminator extends WindowAdapter
- {
- public void windowClosing(WindowEvent e)
- {
- if (user agrees)
- System.exit(0);
- }
- }

- ***java.awt.event.WindowListener***
- void windowOpened(WindowEvent e)
  is called after the window has been opened.
- void windowClosing(WindowEvent e)
  is called when the user has issued a window manager command to close the window. Note that the window will close only if its hide or dispose method is called.
- void windowClosed(WindowEvent e)
  is called after the window has closed.
- void windowIconified(WindowEvent e)
  is called after the window has been iconified.
- void windowDeiconified(WindowEvent e)
  is called after the window has been deiconified.
- void windowActivated(WindowEvent e)
  is called after the window has become active. Only a frame or dialog can be active.
- Typically, the window manager decorates the active window, for example, by highlighting the title bar.
- void windowDeactivated(WindowEvent e)
  is called after the window has become deactivated.

The AWT Event Hierarchy

- The Event Object class has a subclass AWT Event, which is the parent of all AWT event classes.


Event Object
AWT Event
Action Event
Adjustment Event
Component Event
Item Event
Focus Event
Input Event
Paint Event
Window Event
Key Event
Mouse Event
MouseWheel Event

- Here is a list of those AWT Event actually passed to listener
- ActonEvent
- AdjustmentEvent
- ComponenetEvent
- Container Event
- FocusEvent
- ItemEvent
- KeyEvent
- MouseEvent
- TextEvent
- WindowEvent

- The javax.swing.event package contain additional events that specific to swing component .
- There are 11 Listener Interfaces in java.awt.evet package
- ActonListener
- AdjustmentListener
- ComponenetListener
- ContainerListener
- FocusListener
- ItemListener
- KeyListener
- MouseListener
- MouseMotionListener
- TextListener
- WindowListener

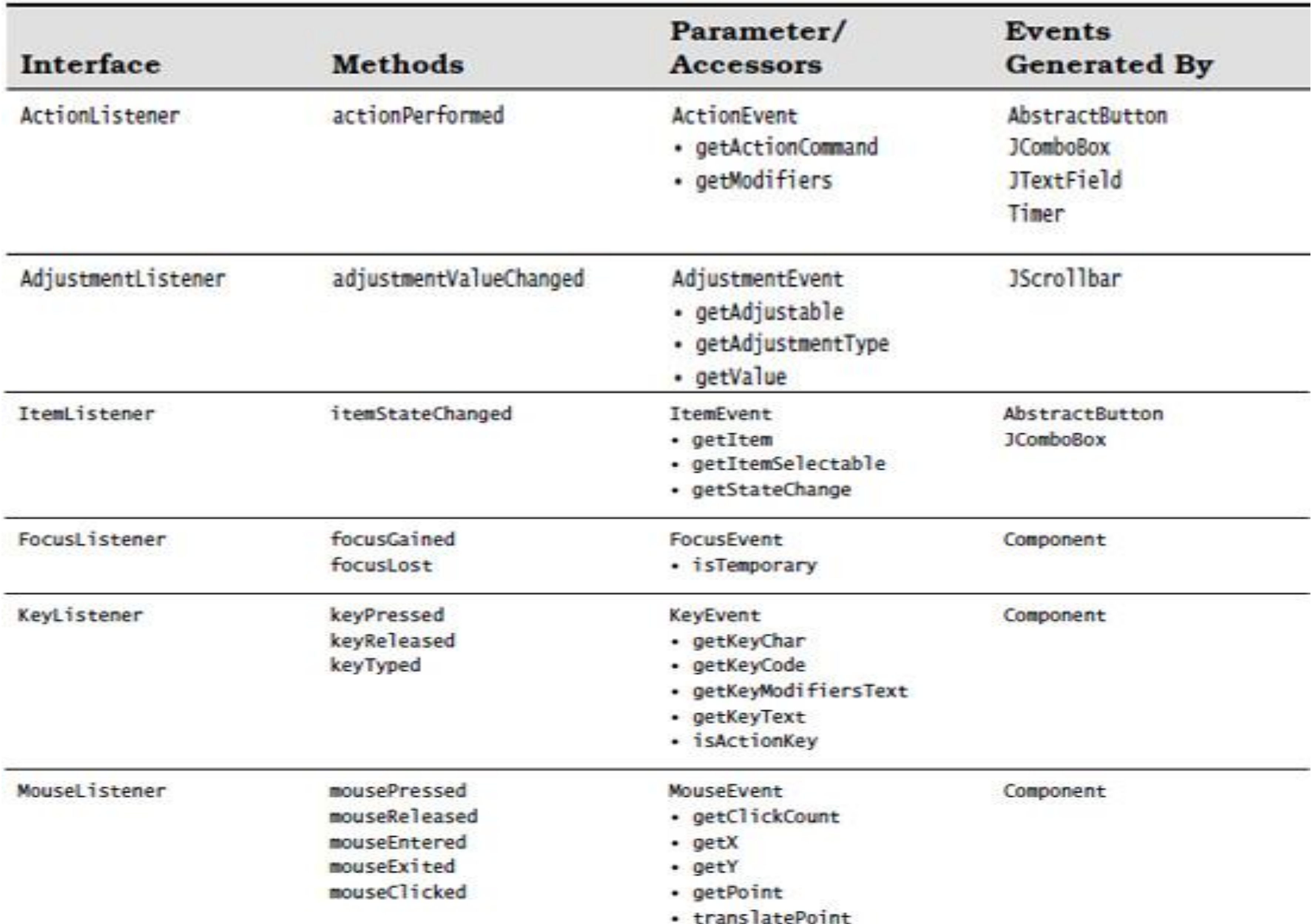
- 7 Adapter Classes
- ComponenetAdapter
- ContainerAdapter
- FocusAdapter
- KeyAdapter
- MouseAdapter
- MouseMotionAdapter
- WindowAdapter

Semantic and Low-Level Events

- The AWT makes a useful distinction between *low - level* and *semantic* events.
- A semantic event is one that expresses what the user is doing, such as “clicking that button”; hence, an Action-Event is a semantic event.
- Low-level events are those events that make this possible. In the case of a button click, this is a mouse down, a series of mouse moves, and a mouse up (but only if the mouse up is inside the button area).
- Or it might be a keystroke, which happens if the user selects the button with the TAB key and then activates it with the space bar. Similarly, adjusting scrollbar is a semantic event, but dragging the mouse is a low-level event.

- Here are the most commonly used **semantic event** classes in the java.awt.event package:
- ActionEvent (for a button click, a menu selection, selecting a list item, or ENTER typedin a text field)
- AdjustmentEvent (the user adjusted a scrollbar)
- ItemEvent (the user made a selection from a set of checkbox or list items)
- **Five low-level** event classes are commonly used:
- KeyEvent (a key was pressed or released)
- MouseEvent (the mouse button was pressed, released, moved, or dragged)
- MouseWheelEvent (the mouse wheel was rotated)
- FocusEvent (a component got focus or lost focus)
- WindowEvent (the window state changed)

- **Event Handling Summary**



Interface	Methods	Parameter/ Accessors	Events Generated By
ActionListener	actionPerformed	ActionEvent • getActionCommand • getModifiers	AbstractButton JComboBox JTextField Timer
AdjustmentListener	adjustmentValueChanged	AdjustmentEvent • getAdjustable • getAdjustmentType • getValue	JScrollbar
ItemListener	itemStateChanged	ItemEvent • getItem • getItemSelectable • getStateChange	AbstractButton JComboBox
FocusListener	focusGained focusLost	FocusEvent • isTemporary	Component
KeyListener	keyPressed keyReleased keyTyped	KeyEvent • getKeyChar • getKeyCode • getKeyModifiersText • getKeyText • isActionKey	Component
MouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	MouseEvent • getClickCount • getX • getY • getPoint • translatePoint	Component

ItemListener	itemStateChanged	ItemEvent • getItem • getItemSelectable • getStateChange	AbstractButton JComboBox
FocusListener	focusGained focusLost	FocusEvent • isTemporary	Component
KeyListener	keyPressed keyReleased keyTyped	KeyEvent • getKeyChar • getKeyCode • getKeyModifiersText • getKeyText • isActionKey	Component
MouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	MouseEvent • getClickCount • getX • getY • getPoint • translatePoint	Component
MouseMotionListener	mouseDragged mouseMoved	MouseEvent	Component
MouseWheelListener	mouseWheelMoved	MouseWheelEvent • getWheelRotation • getScrollAmount	Component
WindowListener	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated	WindowEvent • getWindow	Window

MouseMotionListener	mouseDragged mouseMoved	MouseEvent	Component
MouseWheelListener	mouseWheelMoved	MouseWheelEvent • getWheelRotation • getScrollAmount	Component

Low Level Event Types

- **Focus Events**
- A low-level event which indicates that a Component has gained or lost the input focus. This low-level event is generated by a Component (such as a TextField).
- The event is passed to every FocusListener or FocusAdapter object which registered to receive such events using the Component's addFocusListener method. (FocusAdapter objects implement the FocusListener interface.) Each such listener object gets this FocusEvent when the event occurs.

- **void focusGained(FocusEvent e)**
- Invoked when a component gains the keyboard focus.
- **void focusLost(FocusEvent e)**
- Invoked when a component loses the keyboard focus.

- **void requsetFocus()**
- Moves the focus to this component. The component must be visible for this to happen.

- **void transferFocus()**
- Transfer the focus to the next component in the traversal order.

Keyboard Event

- The Java KeyListener is notified whenever you change the state of key. It is notified against KeyEvent. The KeyListener interface is found in java.awt.event package. When user pushes key, a KEY_PRESSED KeyEvent is genered. When user realeses the key, a KEY_REALEASE KeyEvent is genered.
- Then you trap these events in the **keyPressed** and **keyRealsesd** methods of any class that implement the **KeyListener** interface.
- A third method, **keyTyped,** combines the two it reports on the characters that were genered by the users keystrokes.
- It has three methods.
- **Methods of KeyListener interface**
- The signature of 3 methods found in KeyListener interface are given below:
- **public abstract void** keyPressed(KeyEvent e);
- **public abstract void** keyReleased(KeyEvent e);
- **public abstract void** keyTyped(KeyEvent e);

- Java makes a distinction between characters and **virtual key codes**.
- Virtual key codes are indicated with a prefix of

VK_, such as VK_A or VK_SHIFT. Virtual key codes correspond to keys on the keyboard.

- For example, VK_A denotes the key marked A.
- There is no separate lowercase virtual key code-
- The keyboard does not have separate lowercase keys.

- So, suppose that the user types an uppercase “A” in the usual way, by pressing the SHIFT key along with A key.
- Java reports five events in response to this user action.
- Here are the actions and associated events:
- 1) Pressed the SHIFT key (keyPressed called for VK_Shift)
- 2) Pressed the key A key(keyPressed called for VK_A)
- 3) Typed “A” (keyTyped called for an “A”)
- 4)Released the A key (keyRealeased called for VK_A)
- 5)Released the SHIFT key (keyRealeased called for VK_SHIFT)
- If the user typed a lowercase “a”
- 1) Typed “a” (keyTyped called for an “a”)

- **Methods:-**
- char getKeyChar():- returns the character that the user typed.
- int getKeyCode():- returns the virtual key code of this key event.
- static string getKeyText(int keycode) :- returns a string describing the key code.

Mouse Event

- However, if you want to enable the user to draw with the mouse, you will need to trap mouse move, click, and drag events. In this section, we show you a simple graphics editor application that allows the user to place, move, and erase squares on a canvas .
- When the user clicks a mouse button, three listener methods are called: **mouse Pressed** when the mouse is first pressed, **mouseReleased** when the mouse is released, and, finally, **mouseClicked**. If you are only interested in complete clicks, you can ignore the first two methods.

- By using the getX and getY methods on the MouseEvent argument,
- you can obtain the *x* and *y*-coordinates of the mouse pointer when the mouse was clicked. To distinguish between single, double, and triple (!) clicks, use the **getClickCount** method.
- There are now masks
- BUTTON2_MASK == ALT_MASK
- BUTTON3_MASK == META_MASK
- BUTTON1_DOWN_MASK= The mouse button1 key extended modifier.
- BUTTON2_DOWN_MASK= The mouse button2 extended modifier.
- BUTTON3_DOWN_MASK= The mouse button3 extended modifier.
- SHIFT_DOWN_MASK
- CTRL_DOWN_MASK
- ALT_DOWN_MASK
- ALT_GRAPH_DOWN_MASK
- META_DOWN_MASK

- **Methods:-**
- int getX()
- int getY()
- Point getPoint(): -return the x-(horizontal) and y-(vertical) coordinate, or point where the event happened, using the coordinate system of the source.
- int getClickedCount():- returns the number of consecutive mouse clicks associated with this event.

Actions

- Action Event
- The Java ActionListener is notified whenever you click on the button or menu item. It is notified against ActionEvent. The ActionListener interface is found in java.awt.event package. It has only one method: actionPerformed().
- The actionPerformed() method is invoked automatically whenever you click on the registered component.

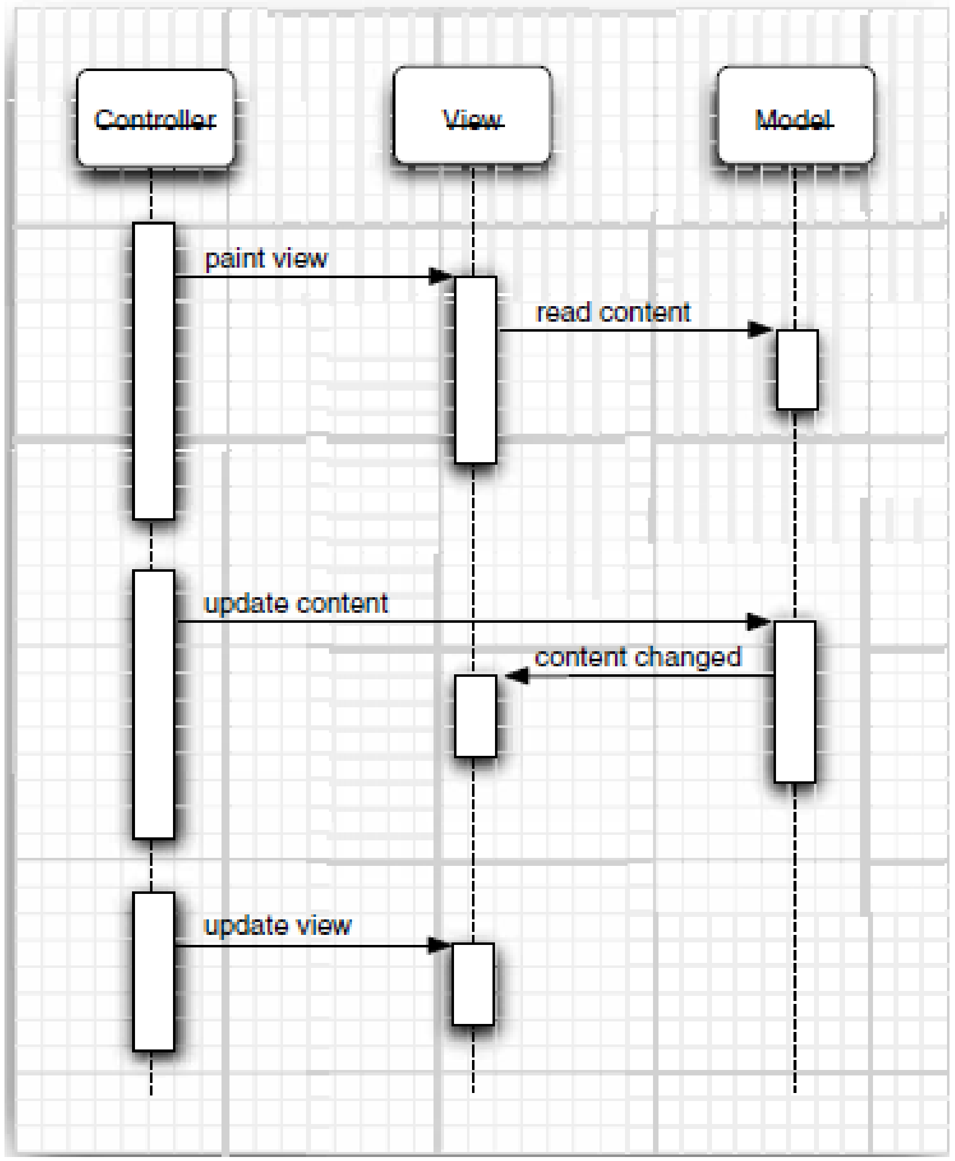
- It is common to have multiple ways to activate the same command. The user can choose a certain function through a menu, a keystroke, or a button on a toolbar. This is easy to achieve in the AWT event model: link all events to the same listener. For example, suppose
- blueAction is an action listener whose actionPerformed method changes the background color to blue. You can attach the same object as a listener to several event sources:
- A toolbar button labeled “Blue”
- A menu item labeled “Blue”
- A keystroke CTRL+B
- Then the color change command is handled in a uniform way, no matter whether it was caused by a button click, a menu selection, or a key press.

The Model View Controller Design Pattern (M-V-C)

- What is Design Patterns MVC Pattern?
- MVC Pattern stands for Model-View-Controller Pattern. This pattern is use to separate application's concerns.
- **Model -** Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.
- **View** - View represents the visualization of the data that model contains.
- **Controller** - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

- Let’s step back for a minute and think about the pieces that make up a user interface component such as a button, a checkbox, a text field, or a sophisticated tree control.
- Every component has three characteristics:
- **Its *content*, such as the state of a button (pushed in or not), or the text in a text field**
- **Its *visual appearance* (color, size, and so on)**
- **Its *behaviour* (reaction to events)**

- The look and feel of the component associated with one object and store the content in *another* object. The model-view-controller (MVC) design pattern teaches how to accomplish this. Implement three separate classes:
- The ***model****,* which stores the content
- The ***view****,* which displays the content
- The ***controller****,* which handles user input

- **Interactions among model, view, and controller objects**
- 

Introduction to Layout Management

- we briefly cover how to arrange these components inside a frame.
- Unlike Visual Basic, the JDK has no form designer. You need to write code to position (lay out) the user interface components where you want them to be.
- **LayoutManagers:-** The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes of layout managers.

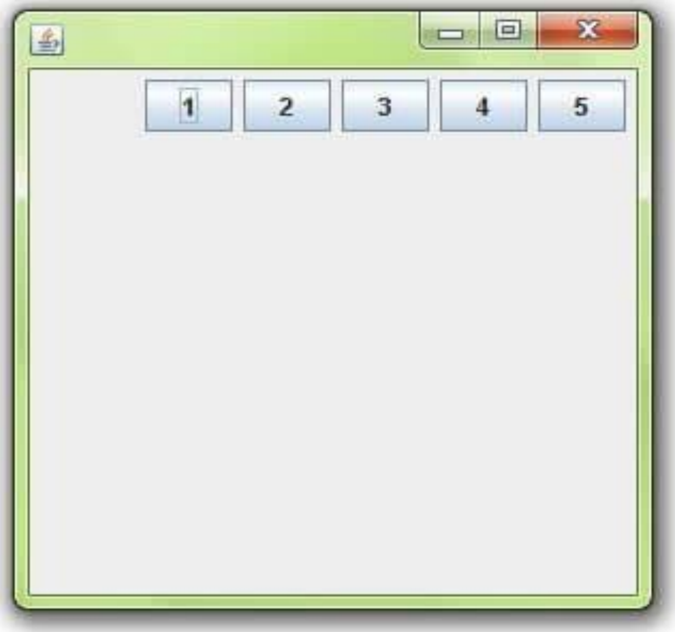
- The layout manager automatically positions all the components within the container. If we do not use layout manager then also the components are positioned by the default layout manager.
- It is possible to layout the controls by hand but it becomes very difficult because of the following two reasons.
- 1) It is very tedious to handle a large number of controls within the container.
- 2) Oftenly the width and height information of a component is not given when we need to arrange them.

- Java provide us with various layout manager to position the controls. The properties like size,shape and arrangement varies from one layout manager to other layout manager.
- The layout manager is associated with every Container object. Each layout manager is an object of the class that implements the LayoutManager interface.anager to other layout manager.

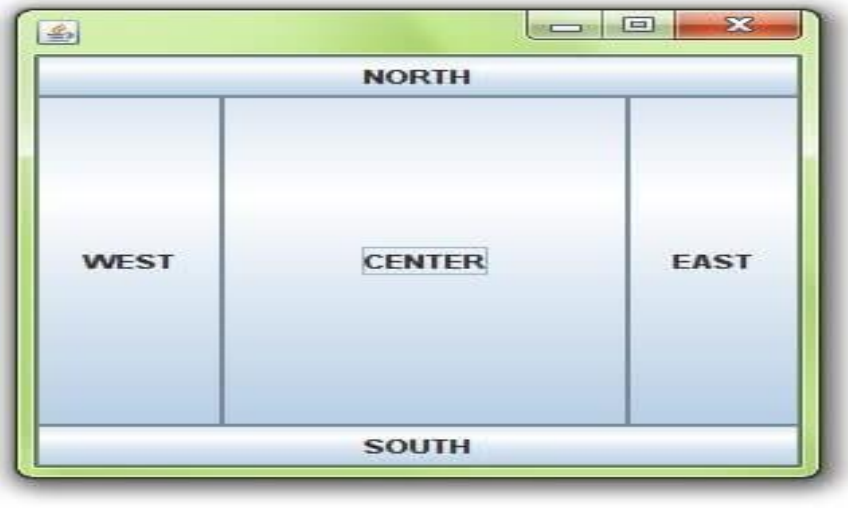
There are 6 layout managers in Java:-

- **FlowLayout**: **Flow layout** is the default layout. The FlowLayout is used to arrange the components in a line, one after another (in a flow). It is the default layout of applet or panel.
- Fields of FlowLayout class
- **public static final int LEFT**
- **public static final int RIGHT**
- **public static final int CENTER**
- **public static final int LEADING**
- **public static final int TRAILING**
- Constructors of FlowLayout class
- **FlowLayout():** creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.
- **FlowLayout(int align):** creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
- **FlowLayout(int align, int hgap, int vgap):** creates a flow layout with the given alignment and the given horizontal and vertical gap.

- **Example of Flow Layout**
- **import** java.awt.*;
- **import** javax.swing.*;
-
- **public class** MyFlowLayout{
- JFrame f;
- MyFlowLayout(){
-     f=**new** JFrame();
-
-     JButton b1=**new** JButton("1");
-     JButton b2=**new** JButton("2");
-     JButton b3=**new** JButton("3");
-     JButton b4=**new** JButton("4");
-     JButton b5=**new** JButton("5");
-
-     f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
-
-     f.setLayout(**new** FlowLayout(FlowLayout.RIGHT));
-     //setting flow layout of right alignment
-
-     f.setSize(300,300);
-     f.setVisible(**true**);
- }
- **public static void** main(String[] args) {
-     **new** MyFlowLayout();
- }
- }



- **BorderLayout**: It arranges all the components along the edges or the middle of the container i.e. **top, bottom, right and left** edges of the area. The components added to the top or bottom gets its preferred height, but its width will be the width of the container and also the components added to the left or right gets its preferred width, but its height will be the remaining height of the container. The components added to the centre gets neither its preferred height or width. It covers the remaining area of the container.



- The BorderLayout provides five constants for each region:
- **public static final int NORTH**
- **public static final int SOUTH**
- **public static final int EAST**
- **public static final int WEST**
- **public static final int CENTER**

- **Example of Border Layout**

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

public class Border {
JFrame f;
Border(){
    f=new JFrame();

    JButton b1=new JButton("NORTH");;
    JButton b2=new JButton("SOUTH");;
    JButton b3=new JButton("EAST");;
    JButton b4=new JButton("WEST");;
    JButton b5=new JButton("CENTER");;

    f.add(b1,BorderLayout.NORTH);
    f.add(b2,BorderLayout.SOUTH);
    f.add(b3,BorderLayout.EAST);
    f.add(b4,BorderLayout.WEST);
    f.add(b5,BorderLayout.CENTER);

    f.setSize(300,300);
    f.setVisible(true);
}
public static void main(String[] args) {
    new Border();
}
}
```

Sophisticated Layout Management

- **GridLayout**: It arranges all the components in a grid of **equally sized cells**, adding them from the **left to righ**t and **top to bottom**. Only one component can be placed in a cell and each region of the grid will have the same size. When the container is resized, all cells are automatically resized. The order of placing the components in a cell is determined as they were added.

- **BoxLayout**: It arranges multiple components in either **vertically or horizontally**, but not both. The components are arranged from **left to right or top to bottom**. If the components are aligned **horizontally**, the height of all components will be the same and equal to the largest sized components. If the components are aligned **vertically**, the width of all components will be the same and equal to the largest width components.

- **GridBagLayout**: It is a powerful layout which arranges all the components in a grid of cells and maintains the aspect ration of the object whenever the container is resized. In this layout, cells may be different in size. It assigns a consistent horizontal and vertical gap among components. It allows us to specify a default alignment for components within the columns or rows.
- **CardLayout**: It arranges two or more components having the same size. The components are **arranged in a deck**, where all the cards of the same size and the **only top card are visible at any time**. The first component added in the container will be kept at the top of the deck. The default gap at the left, right, top and bottom edges are zero and the card components are displayed either **horizontally or vertically.**

TEXT INPUT

- You can use the JTextField and JText- Area components for gathering text input.
- A text field can accept only one line of text; a text area can accept multiple lines of text.
- A JPasswordField accepts one line of text without showing the contents.
- All three of these classes inherit from a class called JTextComponent.
- For example, the methods that get or set the text in a text field or text area are actually methods in JTextComponent.

- ***Text Fields***
- The usual way to add a text field to a window is to add it to a panel or other container— just as you would a button:
- JPanel panel = new JPanel();
- JTextField textField = new JTextField("Default input", 20);
- panel.add(textField);
- This code adds a text field and initializes the text field by placing the string "Default input" inside it.
- The second parameter of this constructor sets the width. In this case, the width is 20 “columns.”

- In general, you want to let the user add text (or edit the existing text) in a text field. Quite often these text fields start out blank.
- To make a blank text field, just leave out the string as a parameter for the JTextField constructor:
- JTextField textField = new JTextField(20);
- You can change the content of the text field at any time by using the setText method from the JText Component parent class
- Eg. textField.setText("Hello!");
- The **getText()** method this method returns the exact text that the user typed.
- To trim any extraneous leading and trailing spaces from the data in a text field, apply the trim method to the return value of getText:
- **String text = textField.getText().trim();**

- To change the font in which the user text appears, use the setFont method.
- JTextField(int cols)
  constructs an empty JTextField with a specified number of columns.
- JTextField(String text, int cols)
  constructs a new JTextField with an initial string and the specified number of columns.
- int getColumns()
- void setColumns(int cols)
  gets or sets the number of columns that this text field should use.
- void revalidate()
  causes the position and size of a component to be recomputed.
- void setFont(Font f)
  sets the font of this component.
- void validate()
  recomputes the position and size of a component. If the component is a container, the positions and sizes of its components are recomputed.
- Font getFont()
  gets the font of this component.

Labels and Labeling Components

- Labels are components that hold text.
- They have no decorations (for example, no boundaries). They also do not react to user input. You can use a label to identify components.
- For example, unlike buttons, text fields have no label to identify them. To label a component that does not itself come with an identifier:
- 1 Construct a JLabel component with the correct text.
- 2 Place it close enough to the component you want to identify so that the user can see that the label identifies the correct component.

- You use constants from the SwingConstants interface to specify alignment.
- That interface defines a number of useful constants such as LEFT, RIGHT, CENTER, NORTH, EAST, and so on.
- JLabel label = new JLabel("User name: ", SwingConstants.RIGHT);
- Or
- JLabel label = new JLabel("User name: ", JLabel.RIGHT);
- JLabel(String text)
- JLabel(Icon icon)
- JLabel(String text, int align)
- JLabel(String text, Icon icon, int align)
  constructs a label.
- String getText()
- void setText(String text)
  gets or sets the text of this label.
- Icon getIcon()
- void setIcon(Icon icon)
  gets or sets the icon of this label

Password Fields

- Password fields are a special kind of text field.
- a password, the characters that the user entered are not actually displayed.
- Instead, each typed character is represented by an *echo character,* typically an asterisk (*).
- Swing supplies a JPasswordField class that implements such a text field.
- The password field is another example of the power of the model - view -controller architecture pattern.
- The password field uses the same model to store the data as a regular text field, but its view has been changed to display all characters as echo characters. • JPassword Field(String text, int columns) constructs a new password field.

- void setEchoChar(char echo)
  sets the echo character for this password field. This is advisory; a particular look and feel may insist on its own choice of echo character. A value of 0 resets the echo character to the default.
- char[] getPassword()
  returns the text contained in this password field. For stronger security, you should overwrite the content of the returned array after use. (The password is not returned as a String because a string would stay in the virtual machine until it is garbage-collected.)

Text Areas

- Sometimes, you need to collect user input that is more than one line long.
- As mentioned earlier, you use the JTextArea component for this collection.
- When you place a text area component in your program, a user can enter any number of lines of text, using the ENTER key to separate them.
- In the constructor for the JTextArea component, you specify the number of rows and columns for the text area. For example,
- textArea = new JTextArea(8, 40); // 8 lines of 40 columns each

- If there is more text than the text area can display, then the remaining text is simply clipped. You can avoid clipping long lines by turning on line wrapping:
- textArea.setLineWrap(true); // long lines are wrapped

Scroll Panes

- In Swing, a text area does not have scrollbars.
- If you want scrollbars, you have to insert the text area inside a *scroll pane.*
- textArea = new JTextArea(8, 40);
- JScrollPane scrollPane = new JScrollPane(textArea);
- The scroll pane now manages the view of the text area.
- Scrollbars automatically appear if there is more text than the text area can display, and they vanish again if text is deleted and the remaining text fits inside the area.
- The scrolling is handled internally in the scroll pane

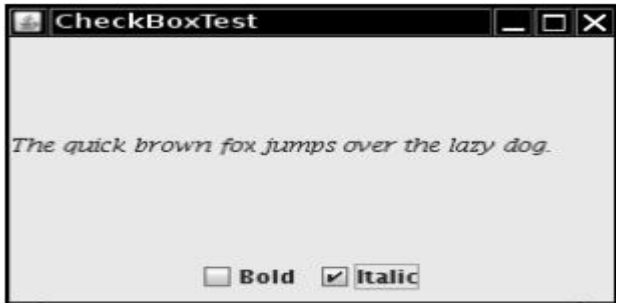
- JTextArea()
- JTextArea(int rows, int cols)
- JTextArea(String text, int rows, int cols)
  constructs a new text area.
- void setColumns(int cols)
  tells the text area the preferred number of columns it should use.
- void setRows(int rows)
  tells the text area the preferred number of rows it should use.
- void append(String newText)
  appends the given text to the end of the text already in the text area.
- void setLineWrap(boolean wrap)
  turns line wrapping on or off.
- void setWrapStyleWord(boolean word)
  If word is true, then long lines are wrapped at word boundaries. If it is false, then long lines are broken without taking word boundaries into account.
- void setTabSize(int c)
  sets tab stops every c columns. Note that the tabs aren’t converted to spaces but cause alignment with the next tab stop.
- JScrollPane(Component c)
  creates a scroll pane that displays the content of the specified component.

CHOICE COMPONENTS

- you would rather give users a finite set of choices than have them enter the data in a text component.
- Using a set of buttons or a list of items tells your users what choices they have. (It also saves you the trouble of error checking.)

Checkboxes

- If you want to collect just a “yes” or “no” input, use a check box component.
- Check boxes automatically come with labels that identify them.
- The user usually checks the box by clicking inside it and turns off the check mark by clicking inside the box again.
- To toggle the check mark, the user can also press the space bar when the focus is in the checkbox.
- a simple program with two checkboxes, one to turn on or off the italic attribute of a font, and the other for bold face.
- Note that the second check box has focus, as indicated by the rectangle around the label.



- bold = new JCheckBox("Bold");
- You use the setSelected method to turn a checkbox on or off. For example:
- bold.setSelected(true);
- The isSelected method then retrieves the current state of each checkbox. It is false if unchecked; true if checked.
- When the user clicks on a checkbox, this triggers an action event. As always, you attach an action listener to the checkbox. In our program, the two checkboxes share the same action listener.

```
ActionListener listener = . . .
bold.addActionListener(listener);
italic.addActionListener(listener);
```

- The action Performed method queries the state of the bold and italic check boxes and sets the font of the panel to plain, bold, italic, or both bold and italic.

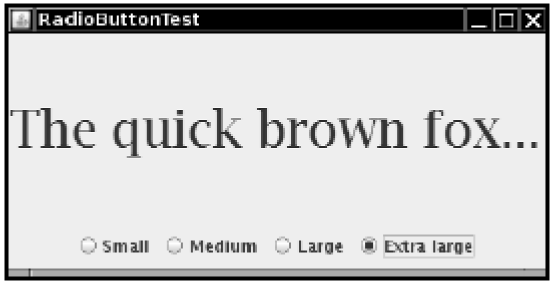
```
 public void actionPerformed(ActionEvent event)
 {
     int mode = 0;
     if (bold.isSelected()) mode += Font.BOLD;
     if (italic.isSelected()) mode += Font.ITALIC;
     label.setFont(new Font("Serif", mode, FONTSIZE));
}
```

- ***javax.swing.JCheckBox 1.2***
- JCheckBox(String label)
- JCheckBox(String label, Icon icon)
  constructs a checkbox that is initially unselected.
- JCheckBox(String label, boolean state)
  constructs a checkbox with the given label and initial state.
- boolean isSelected ()
- void setSelected(boolean state)
  gets or sets the selection state of the checkbox.

Radio Buttons

- In the previous example, the user could check either, both, or neither of the two checkboxes.
- In many cases, we want to require the user to **check only one** of several boxes.
- When another box is checked, the previous box is automatically unchecked.
- Such a group of boxes is often called a *radio button group* because the buttons work like the station selector buttons on a radio.
- When you push in one button, the previously depressed button pops out.

- example. We allow the user to select a font size from among the choices —Small, Medium, Large, and Extra large—but, of course, we will allow the user to select only one size at a time.
- **A radio button group**



Implementing radio button groups is easy in Swing. You construct one object of type Button Group for every group of buttons.
Then, you add objects of type JRadio Button to the button group. The button group object is responsible for turning off the previously set button when a new button is clicked.

```
ButtonGroup group = new ButtonGroup();
JRadioButton smallButton = new
JRadioButton("Small", false);
group.add(smallButton);
JRadioButton mediumButton = new JRadioButton("Medium", true);
group.add(mediumButton);
```

- The second argument of the constructor is true for the button that should be checked initially and false for all others.
- Check boxes are square and contain a check mark when selected. Radio buttons are round and contain a dot when selected.
- When the user checks a radio button, the radio button generates an action event. In our example program, we define an action listener that sets the font size to a particular value:

- ActionListener listener = new
-   ActionListener()
-   {
-   public void actionPerformed(ActionEvent event)
-   {
-   // size refers to the final parameter of the addRadioButton method
-   label.setFont(new Font("Serif", Font.PLAIN, size));
-   }
-   };
- Could we follow the same approach here? We could have a single listener that computes the size as follows:

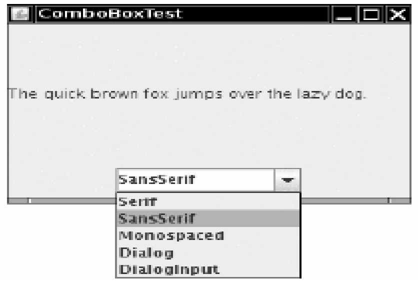
- if (smallButton.isSelected()) size = 8;

- else if (mediumButton.isSelected()) size = 12;

- ***javax.swing.JRadiobutton***
- JRadioButton(String label, Icon icon)
- constructs a radio button that is initially unselected.
- JRadioButton(String label, boolean state)
  constructs a radio button with the given label and initial state.
- void add(AbstractButton b)
- adds the button to the group.
  ButtonModel getSelection()
  returns the button model of the selected button
- String getActionCommand()
  returns the action command for this button model.
- void setActionCommand(String s)
  sets the action command for this button and its model.

Combo Boxes

- If you have more than a handful of alternatives, radio buttons are not a good choice because they take up too much screen space.
- Instead, you can use a combo box. When the user clicks on the component, a list of choices drops down, and the user can then select one of them.



The quick brown fox jumps over the lazy dog.

- If the drop-down list box is set to be *editable*, then you can edit the current selection as if it were a text field. For that reason, this component is called a *combo box*—it combines the flexibility of a text field with a set of predefined choices
- The JComboBox class provides a combo box component.
- You call the **setEditable** method to make the combo box editable. Note that editing affects only the current item.
- You can obtain the current selection or edited text by calling the **getSelectedItem** method.
- You add the choice items with the addItem method.

```
faceCombo = new JComboBox();
faceCombo.setEditable(true);
faceCombo.addItem("Serif");
faceCombo.addItem("SansSerif");
```

- This method adds the string at the end of the list. You can add new items anywhere in the list with the **insertItem**
- **faceCombo.insertItemAt("Monospaced",0);** **At** method:
- If you need to remove items at runtime, you use the removeItem or remove ItemAt method, depending on whether you supply the item to be removed or its position.
- faceCombo.removeItem("Monospaced");
- faceCombo.removeItemAt(0); // remove first item

- ***javax.swing.JComboBox***
- boolean isEditable()
- void setEditable(boolean b)
  gets or sets the editable property of this combo box.
- void addItem(Object item)
  adds an item to the item list.
- void insertItemAt(Object item, int index)
  inserts an item into the item list at a given index.
- void removeItem(Object item)
  removes an item from the item list.
- void removeItemAt(int index)
  removes the item at an index.
- void removeAllItems()
  removes all items from the item list.
- Object getSelectedItem()
  returns the currently selected item.

Sliders

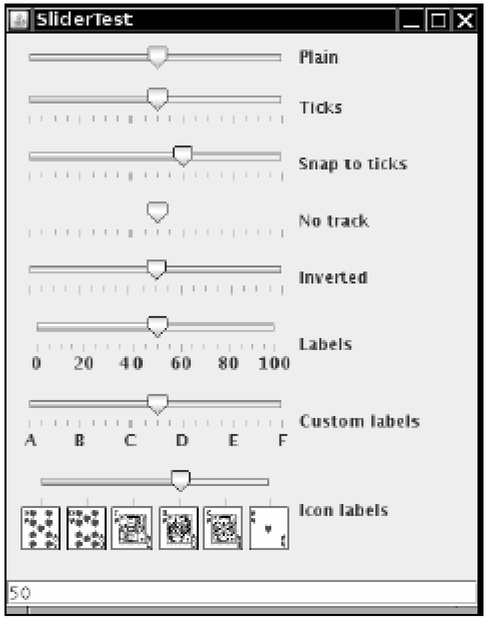
- Combo boxes let users choose from a discrete set of values. Sliders offer a choice from a continuum of values, for example, any number between 1 and 100.
- The most common way of constructing a slider is as follows:

JSlider slider = new JSlider(min, max, initialValue);

- if you want the slider to be vertical, then use the following constructor call:

JSlider slider = new JSlider(SwingConstants.VERTICAL, min, max, initialValue);

- These constructors create a plain slider, such as the top slider. You will see presently how to add decorations to a slider.
- **Sliders**



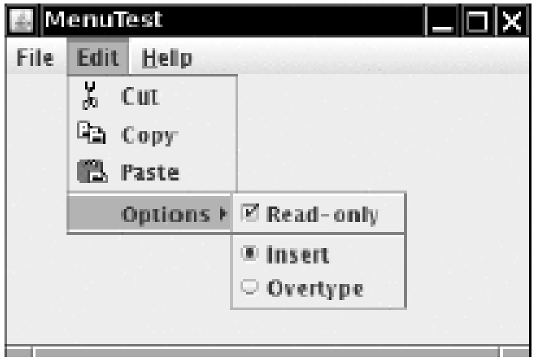
As the user slides the slider bar, the *value* of the slider moves between the minimum and the maximum values. When the value changes, a **ChangeEvent** is sent to all **change listeners.**
To be notified of the change, you call the **addChangeListener** method and install an object that implements the **ChangeListener** interface.
That interface has a single method, **stateChanged**. In that method, you should retrieve the slider value:

- public void stateChanged(ChangeEvent event)
- {
-   JSlider slider = (JSlider) event.getSource();
-   int value = slider.getValue();
-   . . .
-   }

- ***javax.swing.JSlider***
- JSlider()
- JSlider(int direction)
- JSlider(int min, int max)
- JSlider(int min, int max, int initialValue)
- JSlider(int direction, int min, int max, int initialValue) .constructs a horizontal slider with the given direction and minimum, maximum, and initial values.
- void setPaintTicks(boolean b)
  displays ticks if b is true.
- void setMajorTickSpacing(int units)
- void setMinorTickSpacing(int units)
  sets major or minor ticks at multiples of the given slider units.
- void setPaintLabels(boolean b)
  displays tick labels if b is true.
- void setLabelTable(Dictionary table)
  sets the components to use for the tick labels. Each key/value pair in the table has the form new Integer*(value)/component.*
- void setSnapToTicks(boolean b)
  if b is true, then the slider snaps to the closest tick after each adjustment.
- void setPaintTrack(boolean b)
  if b is true, then a track is displayed in which the slider runs.

MENUS

- A ***menu bar*** on top of the window contains the names of the pull-down menus.
- Clicking on a name opens the menu containing *menu items* and *submenus.*
- When the user clicks on a menu item, all menus are closed and a message is sent to the program. a typical menu with a submenu.
- **A menu with a submenu**



- ***Menu Building***
- Building menus is straightforward. You first create a menu bar:
- JMenuBar menuBar = new JMenuBar();
- A menu bar is just a component that you can add anywhere you like. Normally, you want it to appear at the top of a frame. You can add it there with the set JMenuBar method:

frame.setJMenuBar(menuBar);

For each menu, you create a menu object:

JMenu editMenu = new JMenu("Edit");

You add the top-level menus to the menu bar:

menuBar.add(editMenu);

- You add menu items, separators, and submenus to the menu object:

```
JMenuItem pasteItem = new JMenuItem("Paste");
editMenu.add(pasteItem);
editMenu.addSeparator();
JMenu optionsMenu = . . .; // a submenu
editMenu.add(optionsMenu);
```

- You can see separators below the “Paste” and “Read-only” menu items. When the user selects a menu, an action event is triggered. You need to install an action listener for each menu item:

```
ActionListener listener = . . .;
pasteItem.addActionListener(listener);
```

- The method JMenu.add(String s) conveniently adds a menu item to the end of a menu. For example:

```
editMenu.add("Paste");
```

- The add method returns the created menu item, so you can capture it and then add the listener, as follows:

```
JMenuItem pasteItem =  editMenu.add("Paste");
pasteItem.addActionListener(listener);
```

- It often happens that menu items trigger commands that can also be activated through other user interface elements such as toolbar buttons.

- JMenu(String label)
  constructs a menu with the given label.
- JMenuItem add(JMenuItem item)
  adds a menu item (or a menu).
- JMenuItem add(String label)
  adds a menu item with the given label to this menu and returns the item.
- JMenuItem add(Action a)
  adds a menu item with the given action to this menu and returns the item.
- void addSeparator()
  adds a separator line to the menu.
- JMenuItem insert(JMenuItem menu, int index)
  adds a new menu item (or submenu) to the menu at a specific index.
- JMenuItem insert(Action a, int index)
  adds a new menu item with the given action at a specific index.
- void insertSeparator(int index)
  adds a separator to the menu.
- void remove(int index)
- void remove(JMenuItem item)
  removes a specific item from the menu.
- JMenuItem(String label)
  constructs a menu item with a given label.
- JMenuItem(Action a)
  constructs a menu item for the given action.
- void setAction(Action a)
  sets the action for this button or menu item.
- void setJMenuBar(JMenuBar menubar)
  sets the menu bar for this frame.

Icons in Menu Items

- Menu items are very similar to buttons. In fact, the JMenuItem class extends the Abstract Button class. Just like buttons, menus can have just a text label, just an icon, or both. You can specify the icon with the JMenuItem(String, Icon) or JMenuItem(Icon) constructor, or you can set it with the setIcon method that the JMenuItem class inherits from the AbstractButton class.
- Here is an example:
- JMenuItem cutItem = new JMenuItem("Cut", new ImageIcon("cut.gif"));

Checkbox and Radio Button Menu Items

- *Checkbox* and *radio button* menu items display a checkbox or radio button next to the name. When the user selects the menu item, the item automatically toggles between checked and unchecked.
- Apart from the button decoration, you treat these menu items just as you would any others. For example, here is how you create a checkbox menu item:

```
JCheckBoxMenuItem readonlyItem = new
    JCheckBoxMenuItem("Read-only");
optionsMenu.add(readonlyItem);
```

- The radio button menu items work just like regular radio buttons. You must add them to a button group. When one of the buttons in a group is selected, all others are automatically deselected.

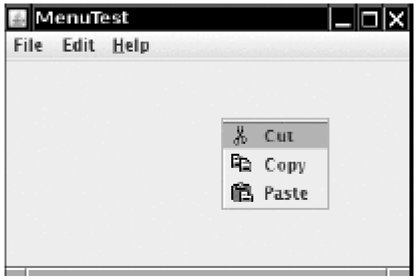
```
ButtonGroup group = new ButtonGroup();
JRadioButtonMenuItem insertItem = new
    JRadioButtonMenuItem("Insert");
insertItem.setSelected(true);
JRadioButtonMenuItem overtypeItem = new
    JRadioButtonMenuItem("Overtype");
group.add(insertItem);
group.add(overtypeItem);
optionsMenu.add(insertItem);
optionsMenu.add(overtypeItem);
```

- With these menu items, you don’t necessarily want to be notified at the exact moment the user selects the item. Instead, you can simply use the isSelected method to test the current state of the menu item.

- JCheckBoxMenuItem(String label)
  constructs the checkbox menu item with the given label.
- JCheckBoxMenuItem(String label, boolean state)
- constructs the checkbox menu item with the given label and the given initial state (true is checked).
- JRadioButtonMenuItem(String label)
  constructs the radio button menu item with the given label.
- JRadioButtonMenuItem(String label, boolean state)
  constructs the radio button menu item with the given label and the given initial state (true is checked).
- boolean isSelected()
- void setSelected(boolean state)

Pop-Up Menus

- A *pop-up menu* is a menu that is not attached to a menu bar but that floats somewhere.
- **A pop-up menu**



- You create a pop-up menu similarly to the way you create a regular menu, but a pop-up menu has no title.

```
JPopupMenu popup = new JPopupMenu();
```

- You then add menu items in the usual way:

```
 JMenuItem item = new  JMenuItem("Cut");
item.addActionListener(listener);
popup.add(item);
```

- Unlike the regular menu bar that is always shown at the top of the frame, you must explicitly display a pop-up menu by using the show method.
- You specify the parent component and the location of the pop-up, using the coordinate system of the parent. For example:
- popup.show(panel, x, y);
- write code to pop up a menu when the user clicks a particular mouse button, the so-called *pop-up trigger.*

- Some systems trigger pop-ups when the mouse button goes down, others when the mouse button goes up.
- void show(Component c, int x, int y)
  shows the pop-up menu.
- boolean isPopupTrigger(MouseEvent event)
  returns true if the mouse event is the pop-up menu trigger.
- boolean isPopupTrigger()
  returns true if this mouse event is the pop-up menu trigger.
- JPopupMenu getComponentPopupMenu()
- void setComponentPopupMenu(JPopupMenu popup)
  gets or sets the pop-up menu for this component.
- boolean getInheritsPopupMenu()
- void setInheritsPopupMenu(boolean b)
  gets or sets the inheritsPopupMenu property. If the property is set and thiscomponent’s pop-up menu is null, it uses its parent’s pop-up menu.

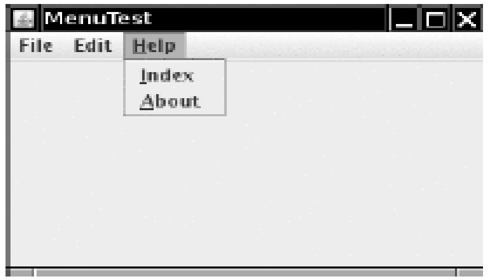
- ***Keyboard Mnemonics and Accelerators:-***
- It is a real convenience for the experienced user to select menu items by *keyboard mnemonics.* You can specify keyboard mnemonics for menu items by specifying a mnemonic letter in the menu item constructor:
- JMenuItem aboutItem = new JMenuItem("About", 'A');
- The keyboard mnemonic is displayed automatically in the menu, with the mnemonic letter underlined.
- For example, in the item defined in the last example, the label will be displayed as “About” with an underlined letter “A”.
- When the menu is displayed, the user just needs to press the A key, and the menu item is selected.

- Sometimes, you don’t want to underline the first letter of the menu item that matches the mnemonic. For example, if you have a mnemonic “A” for the menu item “Save As,” then it makes more sense to underline the second “A” (Save As).
- If you have an Action object, you can add the mnemonic as the value of the

Action .MNEMONIC _KEY key, as follows:

cutAction.putValue(Action.MNEMONIC_KEY, new Integer('A'));

Keyboard mnemonics



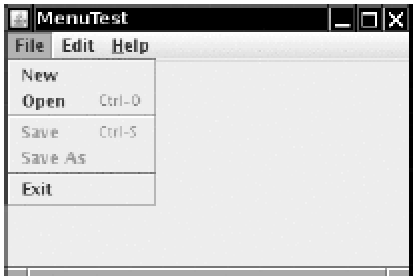
- You can supply a mnemonic letter only in the constructor of a menu item, not in the constructor for a menu. Instead, to attach a mnemonic to a menu, you call the setMnemonic method:
- JMenu helpMenu = new JMenu("Help");
-   helpMenu.setMnemonic('H');
- To select a top-level menu from the menu bar, you press the ALT key together with the mnemonic letter. For example, you press ALT+H to select the Help menu from the menu bar.
- Keyboard mnemonics let you select a submenu or menu item from the currently open menu.

- In contrast, ***accelerators*** are keyboard shortcuts that let you select menu items without ever opening a menu.
- For example, many programs attach the accelerators CTRL+O and CTRL+S to the Open and Save items in the File menu.
- You use the setAccelerator method to attach an accelerator key to a menu item. The setAccelerator method takes an object of type Keystroke.
- For example, the following call attaches the accelerator CTRL+O to the openItem menu item:

openItem.setAccelerator(KeyStroke.getKeyStroke("c
trl O"));

- When the user presses the accelerator key combination, this automatically selects the menu option and fires an action event, as if the user had selected the menu option manually.
- You can attach accelerators only to menu items, not to menus. Accelerator keys don’t actually open the menu. Instead, they directly fire the action event that is associated with a menu.
- when the accelerator is added to a menu item, the key combination is automatically displayed in the menu.

- **Accelerators**



- When the user presses the accelerator key combination, this automatically selects the menu option and fires an action event, as if the user had selected the menu option manually.
- You can attach accelerators only to menu items, not to menus. Accelerator keys don’t actually open the menu. Instead, they directly fire the action event that is associated with a menu.
- when the accelerator is added to a menu item, the key combination is automatically displayed in the menu.

- JMenuItem(String label, int mnemonic)
  constructs a menu item with a given label and mnemonic.
- void setAccelerator(KeyStroke k)
  sets the keystroke k as accelerator for this menu item. The accelerator key is displayed next to the label.
- void setMnemonic(int mnemonic)
  sets the mnemonic character for the button. This character will be underlined in the label.
- void setDisplayedMnemonicIndex(int index) **1.4**
  sets the index of the character to be underlined in the button text. Use this method if you don’t want the first occurrence of the mnemonic character to be underlined.

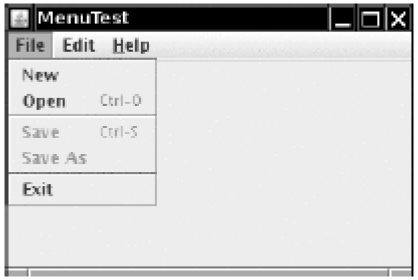
- ***Enabling and Disabling Menu Items***
- Occasionally, a particular menu item should be selected only in certain contexts. For example, when a document is opened for reading only, then the Save menu item is not meaningful. Of course, we could remove the item from the menu with the JMenu.remove method, but users would react with some surprise to menus whose content keeps changing. Instead, it is better to deactivate the menu items that lead to temporarily inappropriate commands. A deactivated menu item is shown in gray, and it cannot be selected.
- To enable or disable a menu item, use the setEnabled method:

```
saveItem.setEnabled(false);
```

- There are two strategies for enabling and disabling menu items.

- you can call setEnabled on the relevant menu items or actions. For example, as soon as a document has been set to read-only mode, you can locate the Save and Save As menu items and disable them. Alternatively, you can disable items just before displaying the menu.
- To do this, you must register a listener for the “menu selected” event. The javax .swing .event package defines a MenuListener interface with three methods:
- void menuSelected(MenuEvent event)
- void menuDeselected(MenuEvent event)
- void menuCanceled(MenuEvent event)
- The menuSelected method is called *before* the menu is displayed. It can therefore be used to disable or enable menu items.
- The following code shows how to disable the Save and Save As actions whenever the Read Only checkbox menu item is selected:

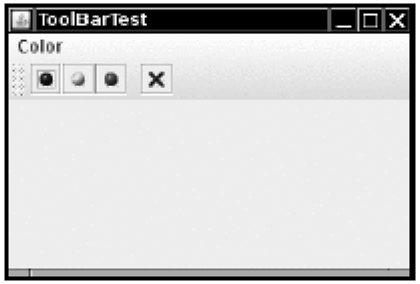
- public void menuSelected(MenuEvent event)
- {
- saveAction.setEnabled(!readonlyItem.isSelected());
- saveAsAction.setEnabled(!readonlyItem.isSelected());
- }
- **Disabled menu items**



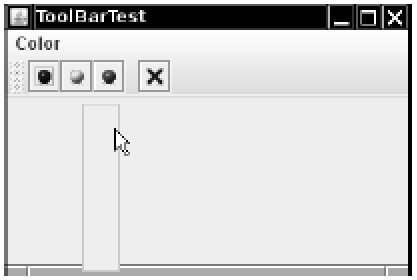
- void setEnabled(boolean b)
  enables or disables the menu item.
- void menuSelected(MenuEvent e)
  is called when the menu has been selected, before it is opened.
- void menuDeselected(MenuEvent e)
  is called when the menu has been deselected, after it has been closed.
- void menuCanceled(MenuEvent e)
  is called when the menu has been canceled, for example, by a user clicking outside the menu.

Toolbars

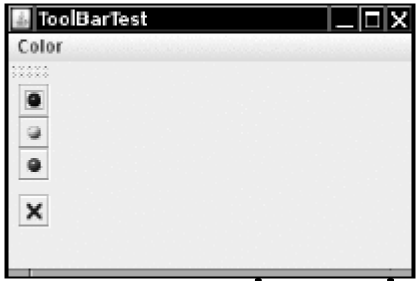
- A toolbar is a button bar that gives quick access to the most commonly used commands in a program .
- **A toolbar**



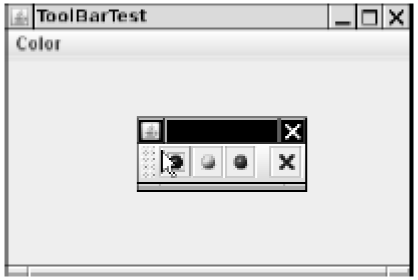
- What makes too s that you can move them elsewhere. You can drag the toolbar to one of the four borders of the frame. When you release the mouse button, the toolbar is dropped into the new location.
- **Dragging the toolbar**



- **The toolbar has been dragged to another border**



- The toolbar can even be completely detached from the frame. A detached toolbar is contained in its own frame.
- When you close the frame containing a detached toolbar, the toolbar jumps back into the original frame.
- **Detaching the toolbar**



- Toolbars are straightforward to program. You add components into the toolbar:

```
JToolBar bar = new JToolBar();
bar.add(blueButton);
```

- The JToolBar class also has a method to add an Action object. Simply populate the toolbar with Action objects, like this:

```
bar.add(blueAction);
```

- The small icon of the action is displayed in the toolbar.
- You can separate groups of buttons with a separator:

```
bar.addSeparator();
```

- For example has a separator between the third and fourth button. Then, you add the toolbar to the frame.

```
add(bar, BorderLayout.NORTH);
```

- You can also specify a title for the toolbar that appears when the toolbar is undocked:

```
bar = new JToolBar(titleString);
```

- By default, toolbars are initially horizontal. To have a toolbar start out as vertical, use

```
bar = new JToolBar(SwingConstants.VERTICAL)
```

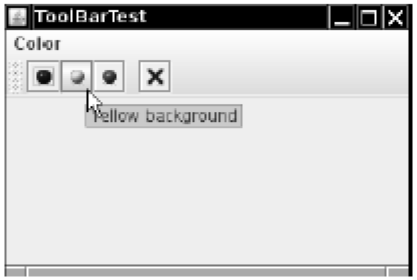
- **or**

```
  bar = new JToolBar(titleString, SwingConstants.VERTICAL)
```

- Buttons are the most common components inside toolbars. But there is no restriction on the components that you can add to a toolbar. For example, you can add a combo box to a toolbar.

Tooltips

- A disadvantage of toolbars is that users are often mystified by the meanings of the tiny icons in toolbars.
- To solve this problem, user interface designers invented *tooltips*.
- A tooltip is activated when the cursor rests for a moment over a button. The tooltip text is displayed inside a colored rectangle. When the user moves the mouse away, the tooltip is removed.
- **A tooltip**



- In Swing, you can add tooltips to any JComponent simply by calling the setToolTipText method:
- exitButton.setToolTipText("Exit");
- Alternatively, if you use Action objects, you associate the tooltip with the SHORT_DESCRIPTION key.

javax.swing.JToolBar

- JToolBar()
- JToolBar(String titleString)
- JToolBar(int orientation)
- JToolBar(String titleString, int orientation)
  constructs a toolbar with the given title string and orientation. orientation is one of Swing Constants .HORIZONTAL (the default) and Swing Constants .VERTICAL.
- JButton add(Action a)
  constructs a new button inside the toolbar with name, icon, short description, and action callback from the given action, and adds the button to the end of the toolbar.
- void addSeparator()
  adds a separator to the end of the toolbar.
- void setToolTipText(String text)
  sets the text that should be displayed as a tooltip when the mouse hovers over the component.

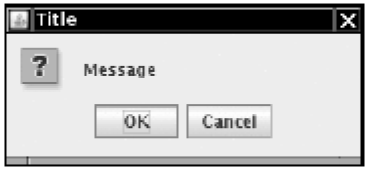
DIALOG BOXES

- This is the most common situation if you write *applets* that run inside a web browser. But if you write applications, you usually want separate dialog boxes to pop up to give information to or get information from the user.
- Just as with most windowing systems, AWT distinguishes between ***modal*** **and** ***modeless*** dialog boxes.
- A modal dialog box won’t let users interact with the remaining windows of the application until he or she deals with it. You use a modal dialog box when you need information from the user before you can proceed with execution.
- For example, when the user wants to read a file, a modal file dialog box is the one to pop up. The user must specify a file name before the program can begin the read operation. Only when the user closes the (modal) dialog box can the application proceed.

- A modeless dialog box lets the user enter information in both the dialog box and the remainder of the application. One example of a modeless dialog is a toolbar.
- The toolbar can stay in place as long as needed, and the user can interact with both the application window and the toolbar as needed.
- Swing has a convenient **JOptionPane** class that lets you put up a simple dialog without writing any special dialog box code.

Option Dialogs

- Swing has a set of ready-made simple dialogs that suffice when you need to ask the user for a single piece of information. The JOptionPane has four static methods to show these simple dialogs:
- showMessageDialog Show a message and wait for the user to click OK
- showConfirmDialog Show a message and get a confirmation (like OK/Cancel)
- showOptionDialog Show a message and get a user option from a set of options
- showInputDialog Show a message and get one line of user input
- Figure below shows a typical dialog. As you can see, the dialog has the following components:
- An icon
- A message
- One or more option buttons
- **An option dialog**



- The input dialog has an additional component for user input. This can be a text field into which the user can type an arbitrary string, or a combo box from which the user can select one item.
- The exact layout of these dialogs, and the choice of icons for standard message types, depend on the pluggable look and feel.
- The icon on the left side depends on one of **five *message types***:

ERROR_MESSAGE

INFORMATION_MESSAGE

WARNING_MESSAGE

QUESTION_MESSAGE

PLAIN_MESSAGE

- The buttons on the bottom depend on the dialog type and the *option type*.
- When calling showMessageDialog and show Input Dialog, you get only a standard set of buttons (OK and OK/ Cancel, respectively). When calling showConfirmDialog, you can choose among four option types:
- DEFAULT_OPTION
- YES_NO_OPTION
- YES_NO_CANCEL_OPTION
- OK_CANCEL_OPTION
- Any other object Apply toString and make a button with the resulting string as label The return values of these functions are as follows:
- showMessageDialog None
- showConfirmDialog An integer representing the chosen option
- showOptionDialog An integer representing the chosen option
- showInputDialog The string that the user supplied or selected

- The showConfirmDialog and showOptionDialog return integers to indicate which button the user chose. For the option dialog, this is simply the index of the chosen option or the value CLOSED_OPTION if the user closed the dialog instead of choosing an option.
- For the confirmation dialog, the return value can be one of the following:
- OK_OPTION
- CANCEL_OPTION
- YES_OPTION
- NO_OPTION
- CLOSED_OPTION

Creating Dialogs

- To implement a dialog box, you extend the JDialog class. This is essentially the same process as extending JFrame for the main window for an application. More precisely:

1 In the constructor of your dialog box, call the constructor of the superclass JDialog.

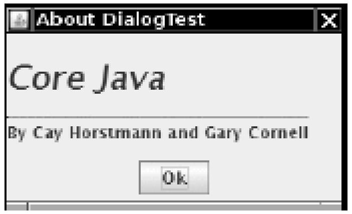
2 Add the user interface components of the dialog box.

3 Add the event handlers.

4 Set the size for the dialog box.

- When you call the superclass constructor, you will need to supply the *owner frame*, the title of the dialog, and the *modality*.
- The owner frame controls where the dialog is displayed. You can supply null as the owner; then, the dialog is owned by a hidden frame.

- **An About dialog box**
- Here’s the code for a dialog box:
- public AboutDialog extends JDialog
- {
- public AboutDialog(JFrame owner)
- {
- super(owner, "About DialogTest", true);
- add(new JLabel("<html><h1><i>Core Java</i></h1><hr>By Cay Horstmann
- and Gary Cornell</html>"),
- BorderLayout.CENTER);
- JPanel panel = new JPanel();
- JButton ok = new JButton("Ok");
- ok.addActionListener(new
- ActionListener()
- {
- public void actionPerformed(ActionEvent event)
- {
- setVisible(false);
- }
- });
- panel.add(ok);
- add(panel, BorderLayout.SOUTH);
- setSize(250, 150);
- }
- }

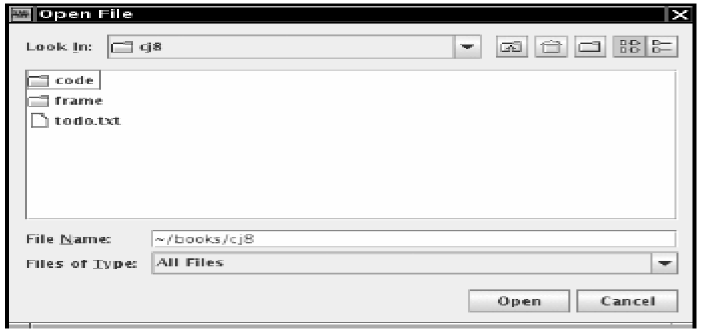


- As you can see, the constructor adds user interface elements: in this case, labels and a button. It adds a handler to the button and sets the size of the dialog. To display the dialog box, you create a new dialog object and make it visible:
- JDialog dialog = new AboutDialog(this);
-   dialog.setVisible(true);
- Actually, in the sample code below, we create the dialog box only once, and we can reuse it whenever the user clicks the About button.
- if (dialog == null) // first time
-   dialog = new AboutDialog(this);
-   dialog.setVisible(true);

- When the user clicks the Ok button, the dialog box should close. This is handled in the event handler of the Ok button:
- ok.addActionListener(new
- ActionListener()
- {
- public void actionPerformed(ActionEvent event)
- {
- setVisible(false);
- }
- });
- When the user closes the dialog by clicking on the Close box, then the dialog is also hidden. Just as with a JFrame, you can override this behavior with the setDefaultCloseOperation method.

File Dialogs

- When you write an application, you often want to be able to open and save files.
- A good file dialog box that shows files and directories and lets the user navigate the file system is hard to write, and you defnitely don’t want to reinvent that wheel.
- Fortunately, Swing provides a JFileChooser class that allows you to display a file dialog box similar to the one that most native applications use.
- JFileChooser dialogs are always modal. Note that the JFileChooser class is not a subclass of JDialog. Instead of calling set- Visible(true), you call showOpenDialog to display a dialog for opening a file or you call show Save Dialog to display a dialog for saving a file.
- The button for accepting a file is then automatically labeled Open or Save. You can also supply your own button label with the showDialog method. example of the file chooser dialog box.
- **File chooser dialog box**



- Here are the steps needed to put up a file dialog box and recover what the user chooses from the box:

1 Make a JFileChooser object. Unlike the constructor for the JDialog class, you do not supply the parent component. This allows you to reuse a file chooser dialog with multiple frames.

- For example:

```
JFileChooser chooser = new JFileChooser();
```

2 Set the directory by calling the setCurrentDirectory method.

- For example, to use the current working directory

```
chooser.setCurrentDirectory(new  File("."));
```

3 If you have a default file name that you expect the user to choose, supply it with the set Selected File method:

```
 chooser.setSelectedFile(new  File(filename));
```

4 To enable the user to select multiple files in the dialog, call the setMultiSelectionEnabled method. This is, of course, entirely optional and not all that common.

```
 chooser.setMultiSelectionEnabled(true);
```

5 If you want to restrict the display of files in the dialog to those of a particular type (for example, all files with extension .gif), then you need to set a *file filter*. We discuss file filters later in this section.

6 By default, a user can select only files with a file chooser. If you want the user to select directories, use the setFileSelectionMode method. Call it with JFile Chooser .FILES _ONLY (the default), JFil eChooser .DIRECTORIES _ONLY, or JFile Chooser.FILES _AND _DIRECTORIES.

7 Show the dialog box by calling the show Open Dialog or show Save Dialog method. You must supply the parent component in these calls:

int result = chooser.showOpenDialog(parent);

or

int result = chooser.showSaveDialog(parent);

The only difference between these calls is the label of the "approve button," the button that the user clicks to finish the file selection. You can also call the showDialog method and pass an explicit text for the approve button:

int result = chooser.showDialog(parent, "Select");

These calls return only when the user has approved, canceled, or dismissed the file dialog. The return value is JFile Chooser .APPROVE _OPTION, JFile Chooser .CANCEL _OPTION, or File Chooser .ERROR _OPTION

8 You get the selected file or files with the getSelectedFile() or getSelectedFiles() method.
These methods return either a single File object or an array of File objects. If you just need the name of the file object, call its getPath method. For example:

```
String filename =
   chooser.getSelectedFile().getPath();
```

Color Choosers

- Many user interface toolkits provide other common dialogs: to choose a date/time, currency value, font, color, and so on. The benefit is twofold.
- Swing provides only one additional chooser, the JColorChooser.
- You use it to let users pick a color value. Like the JFileChooser class, the color chooser is a component, not a dialog, but it contains convenience methods to create dialogs that contain a color chooser component.
- Here is how you show a modal dialog with a color chooser:

- Alternatively, you can display a modeless color chooser dialog. You supply the following:
- A parent component
- The title of the dialog
- A flag to select either a modal or a modeless dialog
- A color chooser
- Listeners for the “OK” and “Cancel” buttons (or null if you don’t want a listener)

- Here is how you make a modeless dialog that sets the background color when the user clicks the OK button:
- chooser = new JColorChooser();
- dialog = JColorChooser.createDialog(
- parent,"Background Color",
- false /* not modal */,
- chooser,
- new ActionListener() // OK button listener
- {
- public void actionPerformed(ActionEvent event)
- {
- setBackground(chooser.getColor());
- }
- },
- null /* no Cancel button listener */);

- You can do even better than that and give the user immediate feedback of the color selection. To monitor the color selections, you need to obtain the selection model of the chooser and add a change listener:
- chooser.getSelectionModel().addChangeListener(new
-   ChangeListener()
-   {
-   public void stateChanged(ChangeEvent event)
-   {do something with chooser.getColor();
-   }
-   });
- In this case, there is no benefit to the OK and Cancel buttons that the color chooser dialog provides. You can just add the color chooser component directly into a modeless dialog:
- dialog = new JDialog(parent, false /* not modal */);
- dialog.add(chooser);
- dialog.pack();

- ***javax.swing.JColorChooser 1.2***
- JColorChooser()
  constructs a color chooser with an initial color of white.
- Color getColor()
- void setColor(Color c)
  gets and sets the current color of this color chooser.
- static Color showDialog(Component parent, String title, Color initialColor)
  shows a modal dialog that contains a color chooser.
- static JDialog createDialog(Component parent, String title, boolean modal, JColorChooser
  chooser, Action Listener okListener, Action Listener cancel Listener) creates a dialog box that contains a color chooser.