10.7.1 AWT

IT1406

Level I - Semester 1

10.7.1.1 Fundamentals of AWT

- **Java AWT** (Abstract Window Toolkit) is an API to develop GUI or window-based applications such as applets in java.
- Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system.
- AWT is heavyweight i.e. its components are using the resources of OS.
- The **java.awt** package provides **classes** for AWT API such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

10.7.1.1 Fundamentals of AWT

AWT Classes

- Some of commonlly used Java AWT classes are given below.

Class	Description
Component	An abstract superclass for various AWT components
Container	A subclass of **Component** that can hold other components
Panel	The simplest concrete subclass of **Container**
Window	Creates a window with no frame, no menu bar, and no title
Frame	Creates a standard window that has a title bar, resize corners, and a menu bar

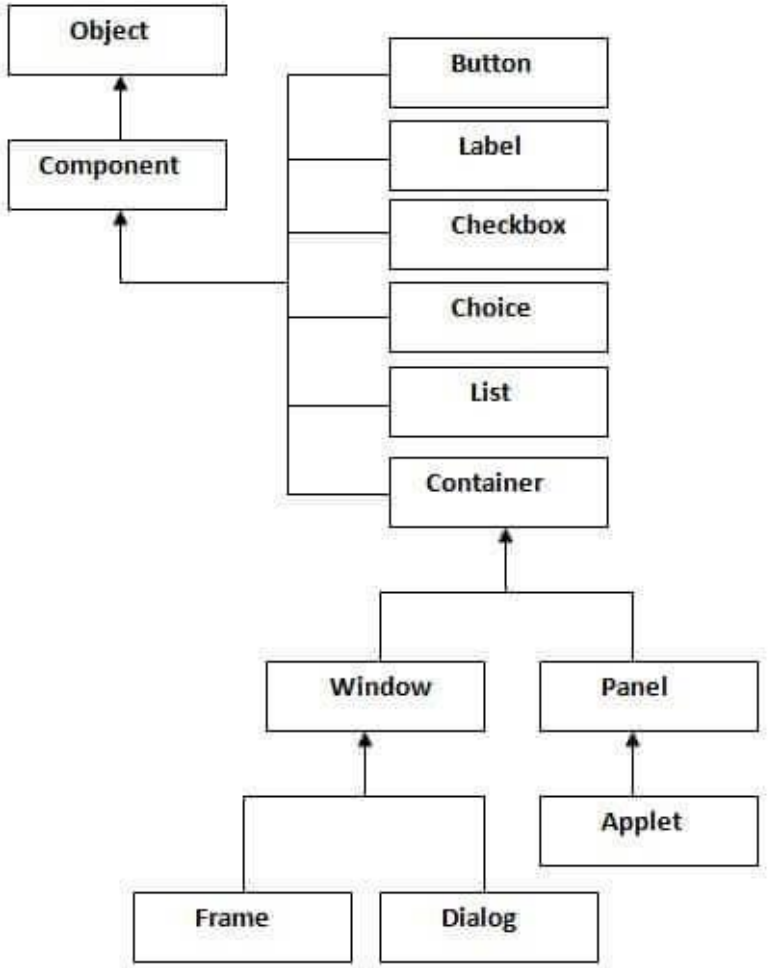
10.7.1.1 Fundamentals of AWT

Class	Description
Canvas	A blank, semantics-free window
Button	Creates a push button control
Image	Encapsulates graphical images
Menu	Creates a pull-down menu
TextArea	Creates a multiline edit control
Scrollbar	Creates a scroll bar control
SystemColor	Contains the colors of GUI widgets such as windows, scroll bars, text, and others

10.7.1.1 Fundamentals of AWT

AWT Hierarchy

- The hierarchy of Java AWT classes are given below.



10.7.1.1 Fundamentals of AWT

Window Fundamentals

- The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level.
- The two most common windows are those derived from **Panel,** which is used by **applets**, and those derived from **Frame,** which creates **a standard application** window. Much of the functionality of these windows is derived from their parent classes.
- Thus, a description of the class hierarchies relating to these two classes is fundamental to their understanding.

10.7.1.1 Fundamentals of AWT

- **Component –** It is an abstract class that encapsulates all of the attributes of a visual component.
- **Container –** It is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container such as Frame, Dialog and Panel
- **Panel –** It is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc
- **Window -** It is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window
- **Frame -** The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc

10.7.1.1 Fundamentals of AWT

How to run an Applet?

- There are two ways to run an applet.
  - By html file.
  - By appletViewer tool (for testing purpose).

Simple example of Applet by html file:

- To execute the applet by html file, create an applet and compile it. After that create an html file and place the applet code in html file.
- Now click the html file.

10.7.1.1 Fundamentals of AWT

- **class** must be **public** because its object is created by Java Plugin software that resides on the browser.

```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{

public void paint(Graphics g){
g.drawString("welcome",150,150);
}

}
```

myapplet.html

```
<html>
<body>
<applet code="First.class" width="300" height="300">
</applet>
</body>
</html>
```

10.7.1.1 Fundamentals of AWT

Simple example of Applet by appletviewer tool

- To execute the applet by appletviewer tool, create an applet that contains applet tag in comment and compile it. After that run it by: appletviewer First.java.

```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{

public void paint(Graphics g){
g.drawString("welcome to applet",150,150);
}

}
/*
<applet code="First.class" width="300" height="300">
</applet>
*/
```

- To execute the applet by appletviewer tool, write in command prompt:

```
c:\>javac First.java
c:\>appletviewer First.java
```

10.7.1.1 Fundamentals of AWT

Creating a frame window in an applet

- Creating a new frame window from within an applet is actually quite easy.
- First, create a subclass of Frame. Next, override any of the standard applet methods, such as **init()**, **start()**, and **stop()** to show or hide the frame as needed.
- Finally, implement the **windowClosing()** method of the WindowListener interface, calling **setVisible(false)** when the window is closed.
- After the applet, the type of window you will most often create is derived from Frame.

10.7.1.1 Fundamentals of AWT

- Frame's constructors are as follows:

  Frame() *throws* HeadlessException

  Frame(String title) *throws* HeadlessException

- The first form creates a standard window that does not contain a title.The second form creates a window with the title specified by title.
- We cannot specify the dimensions of the window. We must set the size of the window after it has been created.
- A **HeadlessException** is thrown if an attempt is made to create a **Frame** instance in an environment that does not support user interaction.
- Some of the methods used when working with windows are as follows:

10.7.1.1 Fundamentals of AWT

Setting the Window's Dimensions

- The **setSize()** method is used to set the dimensions of the window.
- Its signature is shown here:

  void setSize(int *newWidth*, int *newHeight*)
  void setSize(Dimension *newSize*)

- The new size of the window is specified by *newWidth* and *newHeight*, or by the **width** and **height** fields of the **Dimension** object passed in *newSize*.
- The dimensions are specified in terms of pixels.

10.7.1.1 Fundamentals of AWT

- The **getSize()** method is used to obtain the current size of a window.
- Its signature is shown here:

  Dimension getSize()

- This method returns the current size of the window contained within the **width** and **height** fields of a **Dimension** object.

Hiding and Showing a Window

- After a frame window has been created, it will not be visible until you call **setVisible()**. Its signature is shown here:

  void setVisible(boolean *visibleFlag*)

- The component is visible if the argument to this method is **true**. Otherwise, it is hidden.

10.7.1.1 Fundamentals of AWT

Setting a Window's Title

- You can change the title in a frame window using **setTitle()**. Its signature is shownhere:

  void setTitle(String *newTitle*)

- Here, *newTitle* is the new title for the window.

Closing a Frame Window

- When using a frame window, your program must remove that window from the screen when it is closed, by calling **setVisible(false)**.
- To intercept a window-close event, you must implement the **windowClosing()** method of the WindowListener interface.
- Inside **windowClosing()**, you must remove the window from the screen.

10.7.1.1 Fundamentals of AWT

Creating a Frame Window in an Applet

How to create a child frame window from within an applet.....

- Creating a new frame window from within an applet is actually quite easy. The following steps may be used to do it,
  - Create a subclass of **Frame**
  - Override any of the standard window methods,such as **init(),start(),stop(),** and **paint()**.
  - Implement the **windowClosing()** method of the **windowlistener** interface, calling **setVisible(false)** when the window is closed
  - Once you have defined a **Frame** subclass, you can create an object of that class. But it will note be initially visible
  - When created,the window is given a default height and width
  - You can set the size of the window explicitly by calling the **setSize()** method

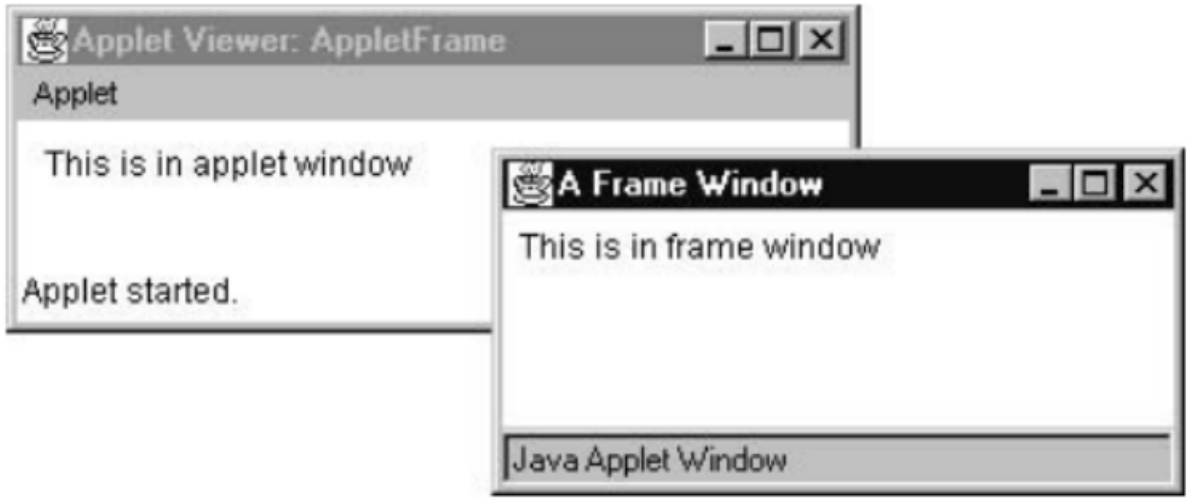
10.7.1.1 Fundamentals of AWT

```
// Create a child frame window from within an applet.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

// Create a subclass of Frame
class SampleFrame extends Frame{
    SampleFrame(String title) {
        super(title);

        // create an object to handle window events
        MyWindowAdapter adapter = new MyWindowAdapter(this);

        // register it to receive those events
        addWindowListener(adapter);
    }

    public void paint(Graphics g) {
        g.drawString("This is in frame window", 10, 40);
    }
}

class MyWindowAdapter extends WindowAdapter {
    SampleFrame sampleFrame;
    public MyWindowAdapter(SampleFrame sampleFrame) {
        this.sampleFrame = sampleFrame;
    }
    public void windowClosing(WindowEvent we) {
        sampleFrame.setVisible(false);
    }
}
```

```
// Create frame window.
public class AppletFrame extends Applet {
    Frame f;
    public void init() {
        f = new SampleFrame("A Frame Window");
        f.setSize(250, 250);
        f.setVisible(true);
    }
    public void start() {
        f.setVisible(true);
    }
    public void stop() {
        f.setVisible(false);
    }
    public void paint(Graphics g) {
        g.drawString("This is in applet window", 10, 20);
    }
}
```

10.7.1.1 Fundamentals of AWT

- Sample Output



10.7.1.1 Fundamentals of AWT

Handling Events in a Frame Window

- Whenever an event occurs in a window, the event handlers defined by that window will be called.
- Each window handles its own events. For example, the following program creates a window that responds to mouse events.
- The main applet window also responds to mouse events.
- When you experiment with this program, you will see that mouse events are sent to the window in which the event occurs.

10.7.1.1 Fundamentals of AWT

```
// Handle mouse events in both child and applet windows.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

// Create a subclass of Frame.
class SampleFrame extends Frame
    implements MouseListener, MouseMotionListener {
        String msg = "";
        int mouseX=10, mouseY=40;
        int movX=0, movY=0;
        SampleFrame(String title) {
            super(title);
            // register this object to receive its own mouse events
            addMouseListener(this);
            addMouseMotionListener(this);
            // create an object to handle window events
            MyWindowAdapter adapter = new MyWindowAdapter(this);
            // register it to receive those events
            addWindowListener(adapter);
        }

    // Handle mouse clicked.
    public void mouseClicked(MouseEvent me) {
    }
    // Handle mouse entered.
    public void mouseEntered(MouseEvent evtObj) {
        // save coordinates
        mouseX = 10;
        mouseY = 54;
        msg = "Mouse just entered child.";
        repaint();
    }
```

```
    // Handle mouse exited.
    public void mouseExited(MouseEvent evtObj) {
        // save coordinates
        mouseX = 10;
        mouseY = 54;
        msg = "Mouse just left child window.";
        repaint();
    }
    // Handle mouse pressed.
    public void mousePressed(MouseEvent me) {
        // save coordinates
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "Down";
        repaint();
    }
    // Handle mouse released.
    public void mouseReleased(MouseEvent me) {
        // save coordinates
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "Up";
        repaint();
    }
    // Handle mouse dragged.
    public void mouseDragged(MouseEvent me) {
        // save coordinates
        mouseX = me.getX();
        mouseY = me.getY();
        movX = me.getX();
        movY = me.getY();
        msg = "*";
        repaint();
    }
```

10.7.1.1 Fundamentals of AWT

```
    // Handle mouse moved.
    public void mouseMoved(MouseEvent me) {
        // save coordinates
        movX = me.getX();
        movY = me.getY();
        repaint(0, 0, 100, 60);
    }
    public void paint(Graphics g) {
        g.drawString(msg, mouseX, mouseY);
        g.drawString("Mouse at " + movX + ", " + movY, 10, 40);
    }
}
class MyWindowAdapter extends WindowAdapter {
    SampleFrame sampleFrame;
    public MyWindowAdapter(SampleFrame sampleFrame) {
        this.sampleFrame = sampleFrame;
    }
    public void windowClosing(WindowEvent we) {
        sampleFrame.setVisible(false);
    }
}
// Applet window.
public class WindowEvents extends Applet
        implements MouseListener, MouseMotionListener {
    SampleFrame f;
    String msg = "";
    int mouseX=0, mouseY=10;
    int movX=0, movY=0;
    // Create a frame window.
    public void init() {
        f = new SampleFrame("Handle Mouse Events");
        f.setSize(300, 200);
        f.setVisible(true);
        // register this object to receive its own mouse events
        addMouseListener(this);
        addMouseMotionListener(this);
    }
```

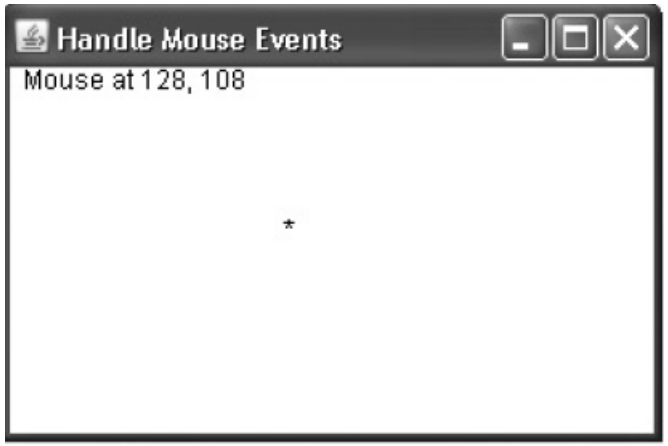


```
// Remove frame window when stopping applet.
public void stop() {
    f.setVisible(false);
}
// Show frame window when starting applet.
public void start() {
    f.setVisible(true);
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
}
// Handle mouse entered.
public void mouseEntered(MouseEvent me) {
    // save coordinates
    mouseX = 0;
    mouseY = 24;
    msg = "Mouse just entered applet window.";
    repaint();
}

// Handle mouse exited.
public void mouseExited(MouseEvent me) {
    // save coordinates
    mouseX = 0;
    mouseY = 24;
    msg = "Mouse just left applet window.";
    repaint();
}
// Handle button pressed.
public void mousePressed(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down";
    repaint();
}
```

10.7.1.1 Fundamentals of AWT

```
    // Handle mouse dragged.
    public void mouseDragged(MouseEvent me) {
        // save coordinates
        mouseX = me.getX();
        mouseY = me.getY();
        movX = me.getX();
        movY = me.getY();
        msg = "*";
        repaint();
    }
    // Handle mouse moved.
    public void mouseMoved(MouseEvent me) {
        // save coordinates
        movX = me.getX();
        movY = me.getY();
        repaint(0, 0, 100, 20);
    }
    // Display msg in applet window.
    public void paint(Graphics g) {
        g.drawString(msg, mouseX, mouseY);
        g.drawString("Mouse at " + movX + ", " + movY, 0, 10);
    }
}
```

Sample Output



10.7.1.1 Fundamentals of AWT

Creating a Windowed Program

- Although creating applets is a common use for Java's AWT, it is also possible to create standalone AWT-based applications.
- To do this, simply create an instance of the window or windows you need inside **main()**.
- For example, the following program creates a frame window that responds to mouse clicks and keystrokes.

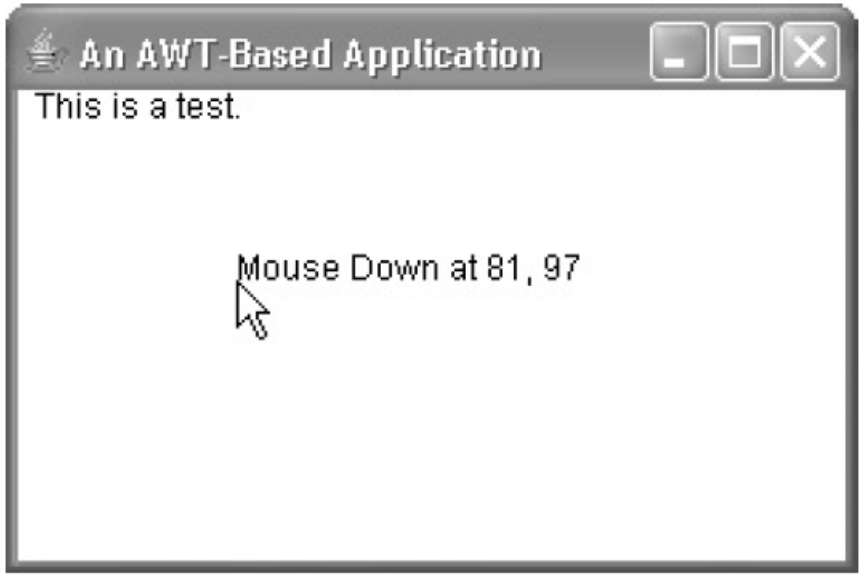
10.7.1.1 Fundamentals of AWT

```
// Create an AWT-based application.
import java.awt.*;
import java.awt.event.*;

// Create a frame window.
public class AppWindow extends Frame {
    String keymsg = "This is a test.";
    String mousemsg = "";
    int mouseX=30, mouseY=30;
    public AppWindow() {
        addKeyListener(new MyKeyAdapter(this));
        addMouseListener(new MyMouseAdapter(this));
        addWindowListener(new MyWindowAdapter());
    }
    public void paint(Graphics g) {
        g.drawString(keymsg, 10, 40);
        g.drawString(mousemsg, mouseX, mouseY);
    }

    // Create the window.
    public static void main(String args[]) {
        AppWindow appwin = new AppWindow();
        appwin.setSize(new Dimension(300, 200));
        appwin.setTitle("An AWT-Based Application");
        appwin.setVisible(true);
    }
}
```

```
class MyKeyAdapter extends KeyAdapter {
    AppWindow appWindow;
    public MyKeyAdapter(AppWindow appWindow) {
        this.appWindow = appWindow;
    }
    public void keyTyped(KeyEvent ke) {
        appWindow.keymsg += ke.getKeyChar();
        appWindow.repaint();
    };
}

class MyMouseAdapter extends MouseAdapter {
    AppWindow appWindow;
    public MyMouseAdapter(AppWindow appWindow) {
        this.appWindow = appWindow;
    }
    public void mousePressed(MouseEvent me) {
        appWindow.mouseX = me.getX();
        appWindow.mouseY = me.getY();
        appWindow.mousemsg = "Mouse Down at " + appWindow.mouseX
            + ", " + appWindow.mouseY;
        appWindow.repaint();
    }
}

class MyWindowAdapter extends WindowAdapter {
    public void windowClosing(WindowEvent we) {
        System.exit(0);
    }
}
```

10.7.1.1 Fundamentals of AWT

Sample Output



- **main()** ends with the call to **appwin.setVisible(true)**.
- However, the program keeps running until close the window.
- When creating a windowed application, **main()** is used to launch its top-level window.

10.7.1.2. Displaying Information Within a Window

- A window is a container for information.
- AWT has ability to present high-quality graphics and text.
- Displaying information within a window comprises of Java's graphics and font handling capabilities.
- First we look at Java's graphics.

Working with Graphics

- The AWT supports a rich set of graphics methods and **Graphics** class contains these methods
- All graphics are drawn relative to a window. The origin of each window is at the top-left corner and is 0,0. Coordinates are specified in pixels.

10.7.1.2. Displaying Information Within a Window

- With Graphics object, we can draw several regular shapes like lines, rectangles, ellipses, circles, ovals and arcs. Any irregular shapes by drawing polygpns.
- **public void paint(Graphics g)** is used to paint the Applet.

Drawing Line

- Lines are drawn by means of the **drawLine()** method. The syntax is

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

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

10.7.1.2. Displaying Information Within a Window

Drawing Rectangles

- The **drawRect()** and **fillRect()** methods display an outlined and filled rectangle, respectively.
- The syntax is

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

  void fillRect(int *left*, int *top*, int *width*, int *height*)
- The upper-left corner of the rectangle is at *left*, *top*. The dimensions of the rectangle are specified by *width* and *height*
- To draw a rounded rectangle, use **drawRoundRect()** or **fillRoundRect()**

10.7.1.2. Displaying Information Within a Window

- The syntax is

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

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

- A rounded rectangle has rounded corners. The upper-left corner of the rectangle is at *left*, *top*. The dimensions of the rectangle are specified by *width* and *height*.
- The diameter of the rounding arc along the X axis is specified by *xDiam*. The diameter of the rounding arc along the Y axis is specified by *yDiam*.

10.7.1.2. Displaying Information Within a Window

Drawing Ellipses and Circles

- To draw an ellipse, use **drawOval()**. To fill an ellipse, use **fillOval()**
- The syntax is

  void drawOval(int *left*, int *top*, int *width*, int *height*)

  void fillOval(int *left*, int *top*, int *width*, int *height*)

- The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by *left*, *top* and whose width and height are specified by *width* and *height*.
- To draw a circle, specify a square as the bounding rectangle.

10.7.1.2. Displaying Information Within a Window

Drawing Arcs

- Arcs can be drawn with **drawArc()** and **fillArc().**
- The syntax is

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

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

- The arc is bounded by the rectangle whose upper-left corner is specified by *left*, *top* and whose width and height are specified by *width* and *height*. The arc is drawn from *startAngle* through the angular distance specified by *sweepAngle*. Angles are specified in degrees.

10.7.1.2. Displaying Information Within a Window

Drawing Polgons

- It is possible to draw arbitrarily shaped figures using **drawPolygon()** and **fillPolygon().**
- The syntax is

  void drawPolygon(int *x* [ ], int *y* [ ], int *numPoints*)

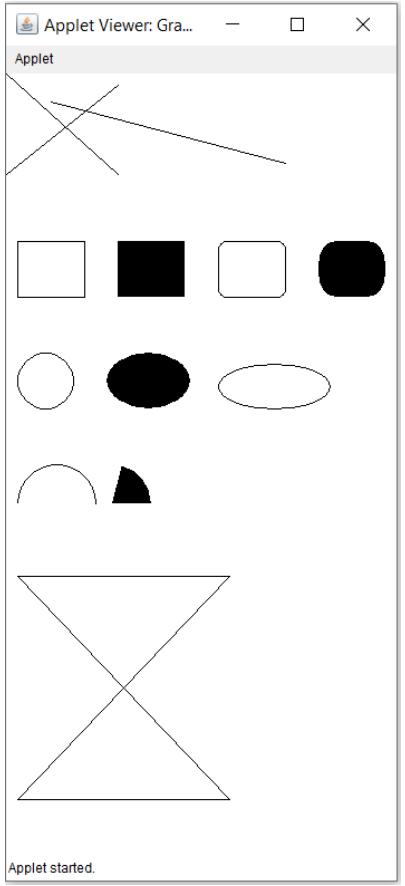
  void fillPolygon(int *x* [ ], int *y* [ ], int *numPoints*)

- The polygon's endpoints are specified by the coordinate pairs contained within the *x* and *y* arrays. The number of points defined by these arrays is specified by *numPoints*.

10.7.1.2. Displaying Information Within a Window

Demonstrating the Drawing Methods

```
import java.awt.*;
import java.applet.*;

public class GraphicsDemo extends Applet {
    public void paint(Graphics g) {
        // Draw lines.
        g.drawLine(0, 0, 100, 90);
        g.drawLine(0, 90, 100, 10);
        g.drawLine(40, 25, 250, 80);

        // Draw rectangles.
        g.drawRect(10, 150, 60, 50);
        g.fillRect(100, 150, 60, 50);
        g.drawRoundRect(190, 150, 60, 50, 15, 15);
        g.fillRoundRect(280, 150, 60, 50, 30, 40);

        // Draw Ellipses and Circles
        g.drawOval(10, 250, 50, 50);
        g.fillOval(90, 250, 75, 50);
        g.drawOval(190, 260, 100, 40);

        // Draw Arcs
        g.drawArc(10, 350, 70, 70, 0, 180);
        g.fillArc(60, 350, 70, 70, 0, 75);

        // Draw a polygon
        int xpoints[] = {10, 200, 10, 200, 10};
        int ypoints[] = {450, 450, 650, 650, 450};
        int num = 5;
        g.drawPolygon(xpoints, ypoints, num);
    }
}
```



Sample Output from the **GraphicsDemo** program

10.7.1.2. Displaying Information Within a Window

Sizing Graphics

- Often, you will want to size a graphics object to fit the current size of the window in which it is drawn.
- To do so, first obtain the current dimensions of the window by calling **getSize()** on the window object.
- It returns the dimensions of the window encapsulated within a Dimension object.
- Once you have the current size of the window, you can scale your graphical output accordingly.

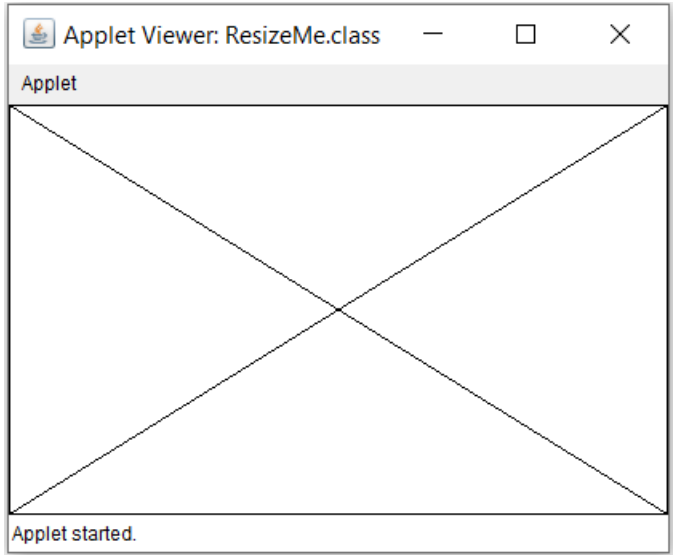
10.7.1.2. Displaying Information Within a Window

- To demonstrate this technique, here is an applet that will start as a 200x200 pixel square and grow by 25 pixels in width and height with each mouse click until the applet gets larger than 500x500.
- At that point, the next click will return it to 200x200, and the process starts over.
- Within the window, a rectangle is drawn around the inner border of the window; within that rectangle, an X is drawn so that it fills the window.
- This applet works in **appletviewer**, but it may not work in a browser window.

10.7.1.2. Displaying Information Within a Window

```
//Resizing output to fit the current size of a window.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code="ResizeMe" width=200 height=200>
</applet>
*/
public class ResizeMe extends Applet {
    final int inc = 25;
    int max = 500;
    int min = 200;
    Dimension d;
    public ResizeMe() {
        addMouseListener(new MouseAdapter() {
        public void mouseReleased(MouseEvent me) {
            int w = (d.width + inc) > max?min :(d.width + inc);
            int h = (d.height + inc) > max?min :(d.height + inc);
            setSize(new Dimension(w, h));
        }
    });
    }

    public void paint(Graphics g) {
        d = getSize();
        g.drawLine(0, 0, d.width-1, d.height-1);
        g.drawLine(0, d.height-1, d.width-1, 0);
        g.drawRect(0, 0, d.width-1, d.height-1);
    }
}
```

Sample Output



10.7.1.2. Displaying Information Within a Window

Working with Color

- Java supports color in a portable, device-independent fashion. The AWT color system allows you to specify any color you want. It then finds the best match for that color, given the limits of the display hardware currently executing your program or applet.
- Thus, your code does not need to be concerned with the differences in the way color is supported by various hardware devices. Color is encapsulated by the **Color** class.

10.7.1.2. Displaying Information Within a Window

Working with Color

- Color defines several constants (for example, **Color.black**) to specify a number of common colors. You can also create your own colors, using one of the color constructors.
- Three commonly used forms are :

  Color(int *red*, int *green*, int *blue*)

  Color(int *rgbValue*)

  Color(float *red*, float *green*, float *blue*)

10.7.1.2. Displaying Information Within a Window

- The first constructor takes three integers that specify the color as a mix of red, green, and blue. These values must be between 0 and 255, as in this example:

```
new Color(255, 100, 100);   // light red
```

- The second color constructor takes a single integer that contains the mix of red, green, and blue packed into an integer. The integer is organized with red in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7. Here is an example of this constructor:

```
int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);
Color darkRed = new Color(newRed);
```

10.7.1.2. Displaying Information Within a Window

- The final constructor, **Color(float, float, float)**, takes three float values (between 0.0 and1.0) that specify the relative mix of red, green, and blue.

- Once you have created a color, you can use it to set the foreground and/or background color by using the **setForeground()** and **setBackground()** methods.

- You can also select it as the current drawing color.

10.7.1.2. Displaying Information Within a Window

Color Methods

- The Color class defines several methods that help manipulate colors. Several are examined here.

Using Hue, Saturation, and Brightness

- The **hue-saturation-brightness (HSB)** color model is an alternative to red-green-blue (RGB) for specifying particular colors.
- Figuratively, **hue** is a wheel of color. The hue can be specified with a number between 0.0 and 1.0, which is used to obtain an angle into the color wheel. (The principal colors are approximately red, orange, yellow, green, blue, indigo, and violet.)

10.7.1.2. Displaying Information Within a Window

- **Saturation** is another scale ranging from 0.0 to 1.0, representing light pastels to intense hues.
- **Brightness** values also range from 0.0 to 1.0, where 1 is bright white and 0 is black.
- Color supplies two methods that let you convert between RGB and HSB. They are shown as follows:

  static int HSBtoRGB(float *hue*, float *saturation*, float *brightness*)

  static float[ ] RGBtoHSB(int *red*, int *green*, int *blue*, float *values*[ ])

10.7.1.2. Displaying Information Within a Window

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

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

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

10.7.1.2. Displaying Information Within a Window

getRed(), getGreen(), getBlue()

- You can obtain the red, green, and blue components of a color independently using **int getRed(),** int **getGreen()** and **int getBlue().**
- Each of these methods returns the RGB color component found in the invoking Color object in the lower 8 bits of an integer.

getRGB()

- To obtain a packed, RGB representation of a colour, use **int getRGB().**

10.7.1.2. Displaying Information Within a Window

Setting the Current Graphics Color

- By default, graphics objects are drawn in the current foreground color. You can change this color by calling the **Graphics** method **void setColor(Colour *newColor*).**

- Here, *newColor* specifies the new drawing color. You can obtain the current color by calling **Color getColor()**.

10.7.1.2. Displaying Information Within a Window

Setting the Current Graphics Color

- By default, graphics objects are drawn in the current foreground color. You can change this color by calling the **Graphics** method **void setColor(Colour *newColor*).**

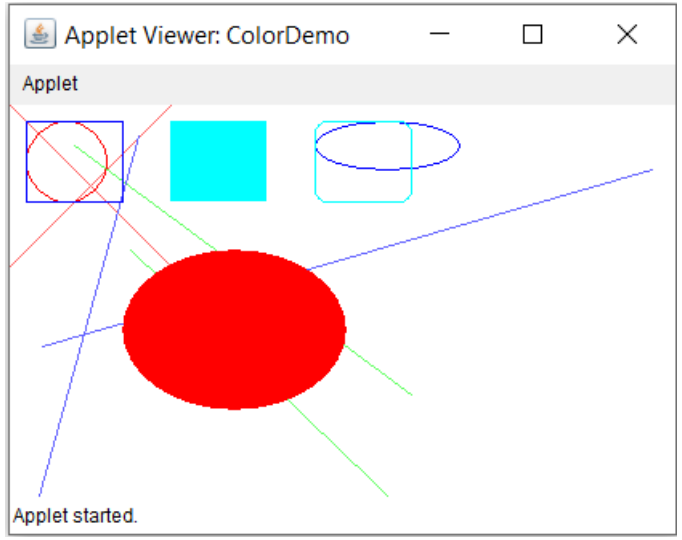
- Here, *newColor* specifies the new drawing color. You can obtain the current color by calling **Color getColor()**.

10.7.1.2. Displaying Information Within a Window

A Color Demonstration Applet

```
import java.awt.*;
import java.applet.*;

public class ColorDemo extends Applet {
    public void paint(Graphics g) {
        Color c1 = new Color(255, 100, 100);
        Color c2 = new Color(100, 255, 100);
        Color c3 = new Color(100, 100, 255);
        g.setColor(c1);
        g.drawLine(0, 0, 100, 100);
        g.drawLine(0, 100, 100, 0);
        g.setColor(c2);
        g.drawLine(40, 25, 250, 180);
        g.drawLine(75, 90, 400, 400);
        g.setColor(c3);
        g.drawLine(20, 150, 400, 40);
        g.drawLine(5, 290, 80, 19);
        g.setColor(Color.red);
        g.drawOval(10, 10, 50, 50);
        g.fillOval(70, 90, 140, 100);
        g.setColor(Color.blue);
        g.drawOval(190, 10, 90, 30);
        g.drawRect(10, 10, 60, 50);
        g.setColor(Color.cyan);
        g.fillRect(100, 10, 60, 50);
        g.drawRoundRect(190, 10, 60, 50, 15, 15);
    }
}
```

Sample Output



10.7.1.2. Displaying Information Within a Window

Setting the Paint Mode

- The *paint mode* determines how objects are drawn in a window. By default, new output to a window overwrites any preexisting contents. However, it is possible to have new objects XORed onto the window by using void **setXORMode(Color *xorColor*).**
- Here, *xorColor* specifies the color that will be XORed to the window when an object is drawn. The advantage of XOR mode is that the new object is always guaranteed to be visible no matter what color the object is drawn over. To return to overwrite mode, call **void setPaintMode().**

10.7.1.2. Displaying Information Within a Window

Working with Fonts

- The AWT supports multiple type fonts and flexibility by abstracting font-manipulation operations and allowing for dynamic selection of fonts.
- Fonts have a family name, a logical font name, and a face name. The *family name* is the general name of the font, such as Courier. The *logical name* specifies a name, such as Monospaced, that is linked to an actual font at runtime. The *face name* specifies a specific font, such as Courier Italic.
- Fonts are encapsulated by the **Font** class.

10.7.1.2. Displaying Information Within a Window

- Fonts are encapsulated by the **Font** class. Several methods and protected variables are defined by Font class.

Method	Description
String getFontName()	Returns the face name of the invoking font
String getName()	Returns the logical name of the invoking font
int getSize()	Returns the size, in points, of the invoking font
int getStyle()	Returns the style values of the invoking font

Variable	Meaning
String name	Name of the font
Float pointSize	Size of the font in points
Int size	Size of the font in points
Int style	Font style

10.7.1.2. Displaying Information Within a Window

Creating and Selecting a Font

- To create a new font, construct a **Font** object that describes that font. One Font constructor has this general form:

  Font(String fontName, int fontStyle, int pointSize)

- Here, fontName specifies the name of the desired font. The name can be specified using either the logical or face name. All Java environments will support Dialog, DialogInput, SansSerif, Serif, and Monospaced fonts.
- Dialog is the font used by your system's dialog boxes. Dialog is also the default if you don't explicitly set a font.

10.7.1.2. Displaying Information Within a Window

- The style of the font is specified by *fontStyle*. It may consist of one or more of these three constants: **Font.PLAIN**, **Font.BOLD**, and **Font.ITALIC**.
- These styles can be combined together. For example, **Font.BOLD | Font.ITALIC** specifies a bold, italics style.
- The size, in points, of the font is specified by *pointSize*. To use a font that you have created, you must select it using **void setFont(Font fontObj)**, which is defined by **Component**. Here *fontObj* is the object that contains the desired font.

10.7.1.2. Displaying Information Within a Window

Managing Text Output Using FontMetrics

- The size of each font may differ and that fonts may be changed while your program is executing, there must be some way to determine the dimensions and various other attributes of the currently selected font.
- For example, to write one line of text after another implies that you have some way of knowing how tall the font is and how many pixels are needed between lines.
- To fill this need, the AWT includes the FontMetrics class, which encapsulates various information about a font.

10.7.1.2. Displaying Information Within a Window

- Following are the common terminology used when describing fonts.

Height	The top-to-bottom size of a line of text
Baseline	The line that the bottoms of characters are aligned to (not counting descent)
Ascent	The distance from the baseline to the top of a character
Descent	The distance from the baseline to the bottom of a character
Leading	The distance between the bottom of one line of text and the top of the next

- **drawString()** method paints a string in the current font and color, beginning at a specified location. However, this location is at the left edge of the baseline of the characters, not at the upper-left corner as is usual with other drawing methods

10.7.1.2. Displaying Information Within a Window

- It is a common error to draw a string at the same coordinate that you would draw a box. For example, if you were to draw a rectangle at coordinate 0,0, you would see a full rectangle. If you were to draw the string “Typesetting” at 0,0, you would only see the tails (or descenders) of the y, p, and g. So that, using font metrics, you can determine the proper placement of each string that you display.
- **FontMetrics** defines several methods that help you manage text output. These methods help you properly display text in a window. Let’s look at some examples.

10.7.1.2. Displaying Information Within a Window

Displaying Multiple Lines of Text

- In general, to display multiple lines of text, your program must manually keep track of the current output position.
- Each time a newline is desired, the Y coordinate must be advanced to the beginning of the next line.
- Each time a string is displayed, the X coordinate must be set to the point at which the string ends.
- This allows the next string to be written so that it begins at the end of the preceding one.

10.7.1.2. Displaying Information Within a Window

- A sample of methods defined by **FontMetrics.**

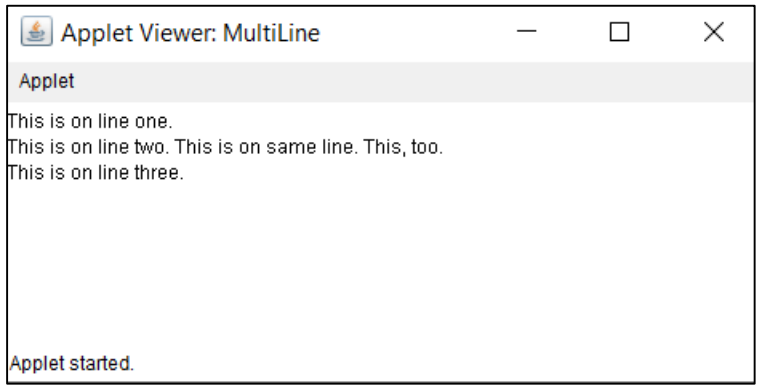
Method	Description
int charWidth(int c)	Returns the width of c
int getAscent()	Returns the ascent of the font
int getHeight()	Returns the height of a line of text. This value can be used to output multiple lines of text in a window
int getLeading()	urns the space between lines of text
Font getFont()	Returns the font

- The following applet shows how to output multiple lines of text in a window. It also displays multiple sentences on the same line. The variables *curX* and *curY* keep track of the current text output position.

10.7.1.2. Displaying Information Within a Window

```
import java.applet.*;
import java.awt.*;

public class MultiLine extends Applet {
    int curX=0, curY=0; // current position
    public void init() {
        Font f = new Font("SansSerif", Font.PLAIN, 12);
        setFont(f);
    }
    public void paint(Graphics g) {
        FontMetrics fm = g.getFontMetrics();
        nextLine("This is on line one.", g);
        nextLine("This is on line two.", g);
        sameLine(" This is on same line.", g);
        sameLine(" This, too.", g);
        nextLine("This is on line three.", g);
        curX = curY = 0; // Reset coordinates for each repaint.
    }
    // Advance to next line.
    void nextLine(String s, Graphics g) {
        FontMetrics fm = g.getFontMetrics();
        curY += fm.getHeight(); // advance to next line
        curX = 0;
        g.drawString(s, curX, curY);
        curX = fm.stringWidth(s); // advance to end of line
    }
    // Display on same line.
    void sameLine(String s, Graphics g) {
        FontMetrics fm = g.getFontMetrics();
        g.drawString(s, curX, curY);
        curX += fm.stringWidth(s); // advance to end of line
    }
}
```

Sample Output

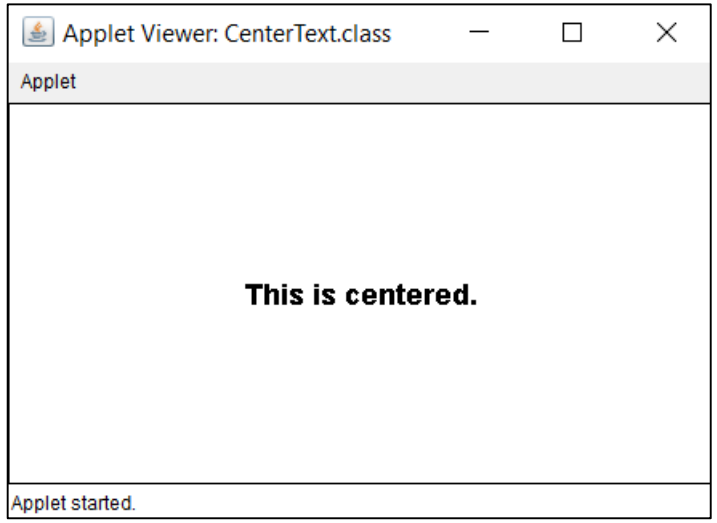


10.7.1.2. Displaying Information Within a Window

Centering Text

- Here is an example that centers text in a window. It obtains the ascent, descent, and width of the string and computes the position at which it must be displayed to be centered.

```
import java.applet.*;
import java.awt.*;

public class CenterText extends Applet {
    final Font f = new Font("SansSerif", Font.BOLD, 18);
    public void paint(Graphics g) {
        Dimension d = this.getSize();
        g.setColor(Color.white);
        g.fillRect(0, 0, d.width,d.height);
        g.setColor(Color.black);
        g.setFont(f);
        drawCenteredString("This is centered.", d.width, d.height, g);
        g.drawRect(0, 0, d.width-1, d.height-1);
    }

    public void drawCenteredString(String s, int w, int h,
    Graphics g) {
        FontMetrics fm = g.getFontMetrics();
        int x = (w - fm.stringWidth(s)) / 2;
        int y = (fm.getAscent() + (h - (fm.getAscent()
        + fm.getDescent()))/2);
        g.drawString(s, x, y);
    }
}
```



Sample Output

10.7.1.3. AWT Controls Fundamentals

- The AWT supports labels, push buttons, check boxes, choice lists, lists, scroll bars and text editing types of controls.
- These controls are subclasses of **Component.**

Adding and Removing Controls

- To include a control in a window, you must add it to the window.
- To do this, you must first create an instance of the desired control and then add it to a window by calling **Component add(Component compRef)**, which is defined by **Container.**
- Here, ***compRef*** is a reference to an instance of the control that you want to add.

10.7.1.3. AWT Controls Fundamentals

- A reference to the object is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.
- Sometimes you will want to remove a control from a window when the control is no longer needed.
- To do this, call **void remove(Component compRef).** This method is also defined by **Container.**
- Here, ***compRef*** *is a reference to the control you want to remove. You can remove all controls* by calling **removeAll().**

10.7.1.3. AWT Controls Fundamentals

Responding to Controls

- Except for labels, which are passive, all other controls generate events when they are accessed by the user.
- For example, when the user clicks on a push button, an event is sent that identifies the push button.
- In general, your program simply implements the appropriate interface and then registers an event listener for each control that you need to monitor.

10.7.1.3. AWT Controls Fundamentals

Labels

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

Label() throws HeadlessException

Label(String str) throws HeadlessException

Label(String str, int how) throws HeadlessException

10.7.1.3. AWT Controls Fundamentals

- The first version creates a blank label.
- The second version creates a label that contains the string specified by *str*. This string is left-justified.
- The third version creates a label that contains the string specified by str using the alignment specified by *how*. The value of *how* must be one of these three constants: Label.LEFT, Label.RIGHT, or Label.CENTER.
- You can set or change the text in a label by using the **void setText(String *str*)** method. You can obtain the current label by calling **String getText()**.
- For setText(), *str* specifies the new label. For getText(), the current label is returned.

10.7.1.3. AWT Controls Fundamentals

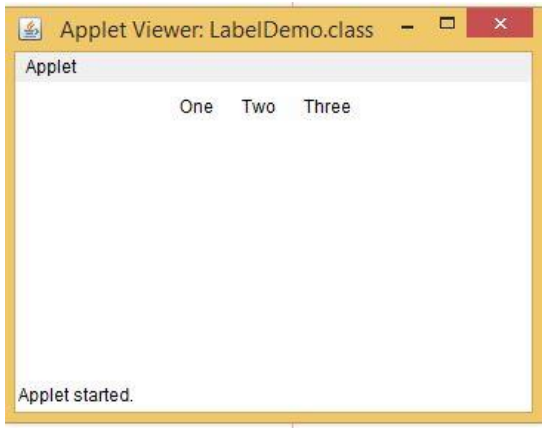
- You can set the alignment of the string within the label by calling void **setAlignment(int *how*).** To obtain the current alignment, call **int getAlignment().** Here, *how* must be one of the alignment constants shown earlier.

- The following example creates three labels and adds them to an applet window:

```
import java.awt.*;
import java.applet.*;

public class LabelDemo extends Applet {
    public void init() {
        Label one = new Label("One");
        Label two = new Label("Two");
        Label three = new Label("Three");
        // add labels to applet window
        add(one);
        add(two);
        add(three);
    }
}
```

10.7.1.3. AWT Controls Fundamentals

- Here is sample output from the **LabelDemo** applet. The labels are organized in the window by the default layout manager.



Using Buttons

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

  Button() throws HeadlessException

  Button(String *str*) throws HeadlessException

10.7.1.3. AWT Controls Fundamentals

- The first version creates an empty button. The second creates a button that contains *str as* a label.
- After a button has been created, you can set its label by calling **void setLabel()**. You can retrieve its label by calling **String getLabel()**. Here, *str* becomes the new label for the button.

Handling Buttons

- Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component.

10.7.1.3. AWT Controls Fundamentals

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

10.7.1.3. AWT Controls Fundamentals

- Here is an example that creates three buttons labeled "Yes", "No", and "Undecided". Each time one is pressed, a message is displayed that reports which button has been pressed.

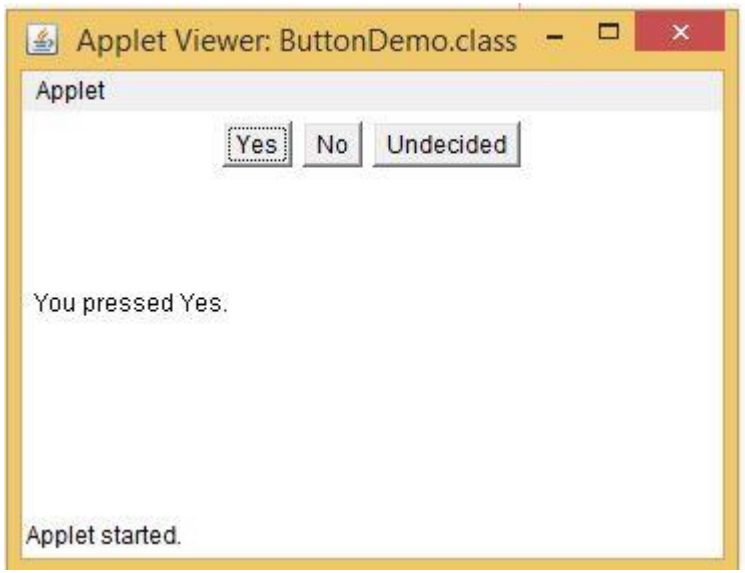
- In this version, the action command of the button (which, by default, is its label) is used to determine which button has been pressed.

- The label is obtained by calling the **getActionCommand()** method on the **ActionEvent object** passed to **actionPerformed().**

10.7.1.3. AWT Controls Fundamentals

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

public class ButtonDemo extends Applet implements ActionListener {
    String msg = "";
    Button yes, no, maybe;
    public void init() {
        yes = new Button("Yes");
        no = new Button("No");
        maybe = new Button("Undecided");
        add(yes);
        add(no);
        add(maybe);
        yes.addActionListener(this);
        no.addActionListener(this);
        maybe.addActionListener(this);
    }

    public void actionPerformed(ActionEvent ae) {
        String str = ae.getActionCommand();
        if(str.equals("Yes")) {
            msg = "You pressed Yes.";
        }
        else if(str.equals("No")) {
            msg = "You pressed No.";
        }
        else {
            msg = "You pressed Undecided.";
        }
        repaint();
    }
    public void paint(Graphics g) {
        g.drawString(msg, 6, 100);
    }
}
```

Sample Output



10.7.1.3. AWT Controls Fundamentals

- As mentioned, in addition to comparing button action command strings, you can also determine which button has been pressed by comparing the object obtained from the **getSource()** method to the button objects that you added to the window.

- To do this, you must keep a list of the objects when they are added.

- The following applet shows this approach:

10.7.1.3. AWT Controls Fundamentals

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

public class ButtonList extends Applet implements ActionListener {
    String msg = "";
    Button bList[] = new Button[3];
    public void init() {
        Button yes = new Button("Yes");
        Button no = new Button("No");
        Button maybe = new Button("Undecided");
        // store references to buttons as added
        bList[0] = (Button) add(yes);
        bList[1] = (Button) add(no);
        bList[2] = (Button) add(maybe);
        // register to receive action events
        for(int i = 0; i < 3; i++) {
            bList[i].addActionListener(this);
        }
    }
    public void actionPerformed(ActionEvent ae) {
        for(int i = 0; i < 3; i++) {
            if(ae.getSource() == bList[i]) {
                msg = "You pressed " + bList[i].getLabel();
            }
        }
        repaint();
    }
    public void paint(Graphics g) {
        g.drawString(msg, 6, 100);
    }
}
```

- This program stores each button reference in an array when the buttons are added to the applet window.
- Inside actionPerformed(), this array is then used to determine which button has been pressed.

10.7.1.3. AWT Controls Fundamentals

Applying Check Boxes

- A check box is a control that is used to turn an option on or off.
- It consists of a small box that can either contain a check mark or not.
- There is a label associated with each check box that describes what option the box represents.
- You change the state of a check box by clicking on it.
- Check boxes can be used individually or as part of a group. Check boxes are objects of the **Checkbox** class.

10.7.1.3. AWT Controls Fundamentals

- Checkbox supports these constructors:

  Checkbox() throws HeadlessException

  Checkbox(String *str) throws HeadlessException*

  Checkbox(String *str, boolean on) throws HeadlessException*

  Checkbox(String *str, boolean on, CheckboxGroup cbGroup) throws HeadlessException*

  Checkbox(String *str, CheckboxGroup cbGroup, boolean on) throws HeadlessException*

- The first form creates a check box whose label is initially blank. The state of the check box is unchecked.
- The second form creates a check box whose label is specified by str. The state of the check box is unchecked.

10.7.1.3. AWT Controls Fundamentals

- The third form allows you to set the initial state of the check box. If *on* is true, the check box is initially checked; otherwise, it is cleared.
- The fourth and fifth forms create a check box whose label is specified by *str* and whose group is specified by *cbGroup*.
- If this check box is not part of a group, then *cbGroup* must be null. The value of on determines the initial state of the check box.
- To retrieve the current state of a check box, call **boolean getState().** To set its state**,** call **void setState(boolean *on*).** You can obtain the current label associated with a check box by calling **String getLabel().** To set the label, call **void setLabel(String *str*).**

10.7.1.3. AWT Controls Fundamentals

- Here, if *on* is **true,** the box is checked**.** If it is **false,** the box is cleared. The string passed in str becomes the new label associated with the invoking check box.

Handling Check Boxes

- Each time a check box is selected or deselected, an item event is generated.
- This is sent to any listeners that previously registered an interest in receiving item event notifications from that component.
- Each listener implements the **ItemListener** interface. That interface defines the **itemStateChanged()** method.

10.7.1.3. AWT Controls Fundamentals

- An **ItemEvent** object is supplied as the argument to this method. It contains information about the event (for example, whether it was a selection or deselection).

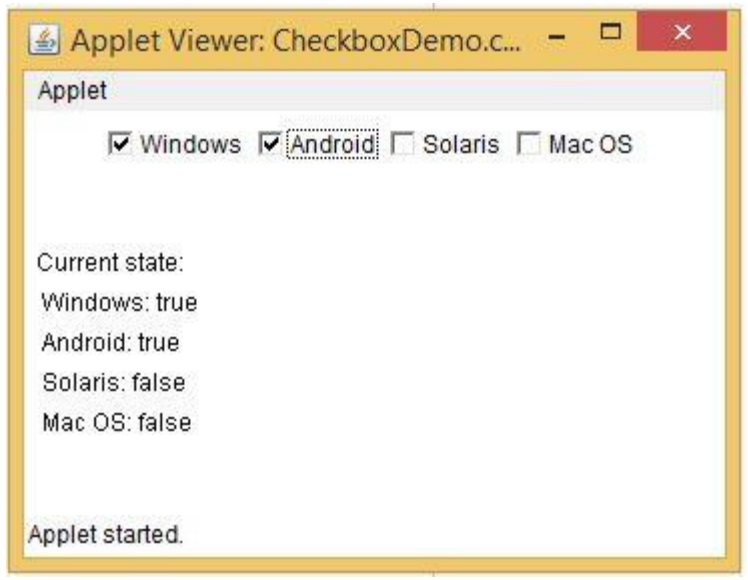
- The following program creates four check boxes. The initial state of the first box is checked. The status of each check box is displayed.

- Each time you change the state of a check box, the status display is updated.

10.7.1.3. AWT Controls Fundamentals

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class CheckboxDemo extends Applet implements ItemListener {
    String msg = ""; Checkbox windows, android, solaris, mac;
    public void init() {
        windows = new Checkbox("Windows", null, true);
        android = new Checkbox("Android");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");
        add(windows); add(android);
        add(solaris); add(mac);
        windows.addItemListener(this);
        android.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }
    // Display current state of the check boxes.
    public void paint(Graphics g) {
        msg = "Current state: ";
        g.drawString(msg, 6, 80);
        msg = " Windows: " + windows.getState();
        g.drawString(msg, 6, 100);
        msg = " Android: " + android.getState();
        g.drawString(msg, 6, 120);
        msg = " Solaris: " + solaris.getState();
        g.drawString(msg, 6, 140);
        msg = " Mac OS: " + mac.getState();
        g.drawString(msg, 6, 160);
    }
}
```

Sample Output



10.7.1.3. AWT Controls Fundamentals

CheckboxGroup

- It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time.
- These check boxes are often called **radio buttons**, because they act like the station selector on a car radio; only one station can be selected at any one time.
- To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes.

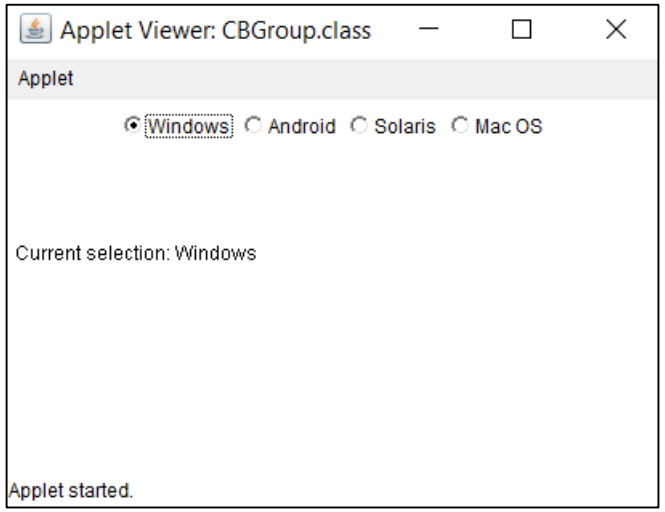
10.7.1.3. AWT Controls Fundamentals

- Check box groups are objects of type **CheckboxGroup.**
- Only the default constructor is defined, which creates an empty group.
- You can determine which check box in a group is currently selected by calling Checkbox **getSelectedCheckbox().**
- You can set a check box by calling **void setSelectedCheckbox(Checkbox *which*).** Here, *which* is the check box that you want to be selected. The previously selected check box will be turned off.
- Here is a program that uses check boxes that are part of a group and the check boxes are circular in shape.

10.7.1.3. AWT Controls Fundamentals

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

public class CBGroup extends Applet implements ItemListener {
    String msg = "";
    Checkbox windows, android, solaris, mac;
    CheckboxGroup cbg;
    public void init() {
        cbg = new CheckboxGroup();
        windows = new Checkbox("Windows", cbg, true);
        android = new Checkbox("Android", cbg, false);
        solaris = new Checkbox("Solaris", cbg, false);
        mac = new Checkbox("Mac OS", cbg, false);
        add(windows);
        add(android);
        add(solaris);
        add(mac);
        windows.addItemListener(this);
        android.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }
    // Display current state of the check boxes.
    public void paint(Graphics g) {
        msg = "Current selection: ";
        msg += cbg.getSelectedCheckbox().getLabel();
        g.drawString(msg, 6, 100);
    }
}
```

Sample Output



10.7.1.3. AWT Controls Fundamentals

Choice Controls

- The **Choice** class is used to create a pop-up list of items from which the user may choose. Thus, a Choice control is a form of menu.
- When inactive, a Choice component takes up only enough space to show the currently selected item.
- When the user clicks on it, the whole list of choices pops up, and a new selection can be made.
- Each item in the list is a string that appears as a left-justified label in the order it is added to the Choice object.

10.7.1.3. AWT Controls Fundamentals

- Choice defines only the default constructor, which creates an empty list.
- To add a selection to the list, call **void add(String *name*).** Here, *name* is the name of the item being added. Items are added to the list in the order in which calls to **add()** occur.
- To determine which item is currently selected, you may call either **String getSelectedItem()** or **int getSelectedIndex().**
- The **getSelectedItem()** method returns a string containing the name of the item. **getSelectedIndex()** returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected.

10.7.1.3. AWT Controls Fundamentals

- To obtain the number of items in the list, call **int getItemCount().**
- You can set the currently selected item using the **void select(int *index*) or void select(String *name*)** method with either a zero-based integer index or a string that will match a name in the list.
- Given an index, you can obtain the name associated with the item at that index by calling **String getItem(int** *index*). Here, *index* specifies the index of the desired item.

Handling Choice Lists

- Each time a choice is selected, an item event is generated.

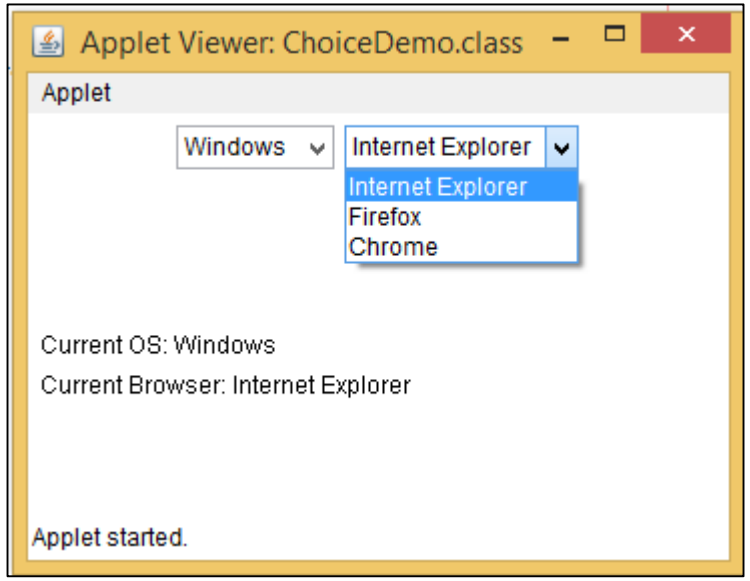
10.7.1.3. AWT Controls Fundamentals

- This is sent to any listeners that previously registered an interest in receiving item event notifications from that component.
- Each listener implements the **ItemListener interface.** That interface defines the **itemStateChanged()** method.
- An **ItemEvent** object is supplied as the argument to this method.
- Here is an example that creates two Choice menus. One selects the operating system. The other selects the browser.

10.7.1.3. AWT Controls Fundamentals

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

public class ChoiceDemo extends Applet implements ItemListener {
    Choice os, browser;
    String msg = "";
    public void init() {
        os = new Choice();
        browser = new Choice();
        // add items to os list
        os.add("Windows"); os.add("Android");
        os.add("Solaris"); os.add("Mac OS");
        // add items to browser list
        browser.add("Internet Explorer"); browser.add("Firefox");
        browser.add("Chrome");
        // add choice lists to window
        add(os);
        add(browser);
        // register to receive item events
        os.addItemListener(this);
        browser.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }
    // Display current selections.
    public void paint(Graphics g) {
        msg = "Current OS: ";
        msg += os.getSelectedItem();
        g.drawString(msg, 6, 120);
        msg = "Current Browser: ";
        msg += browser.getSelectedItem();
        g.drawString(msg, 6, 140);
    }
}
```

Sample Output



10.7.1.3. AWT Controls Fundamentals

Using Lists

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

  List() throws HeadlessException

  List(int *numRows*) throws HeadlessException

  List(int *numRows*, boolean *multipleSelect*) throws HeadlessException

10.7.1.3. AWT Controls Fundamentals

- The first version creates a **List** control that allows only one item to be selected at any one time.
- In the second form, the value of *numRows* specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed).
- In the third form, if *multipleSelect* is **true**, then the user may select two or more items at a time. If it is **false**, then only one item may be selected.
- To add a selection to the list, call add(). It has the following two forms:

  void add(String *name*)

  void add(String *name*, int *index*).

10.7.1.3. AWT Controls Fundamentals

- Here, *name* is the name of the item added to the list.
- The first form adds items to the end of the list. The second form adds the item at the index specified by *index*. Indexing begins at zero. You can specify –1 to add the item to the end of the list.
- For lists that allow only single selection, you can determine which item is currently selected by calling either **String getSelectedItem()** or **int getSelectedIndex().**
- The **getSelectedItem()** method returns a string containing the name of the item. If more than one item is selected, or if no selection has yet been made, **null** is returned.

10.7.1.3. AWT Controls Fundamentals

- The **getSelectedIndex()** returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, –1 is returned.
- For lists that allow multiple selection, you must use either **String[] getSelectedItems()** or **int[] getSelectedIndexes()** to determine the current selections.
- The **getSelectedItems()** returns an array containing the names of the currently selected items. The **getSelectedIndexes()** returns an array containing the indexes of the currently selected items.

10.7.1.3. AWT Controls Fundamentals

- To obtain the number of items in the list, call **int getItemCount().** You can set the currently selected item by using the **void select(int *index*)** method with a zero-based integer index.
- Given an index, you can obtain the name associated with the item at that index by calling **String getItem(int index).** Here, *index* specifies the index of the desired item.

Handling Lists

- To process list events, you will need to implement the **ActionListener** interface. Each time a **List** item is double-clicked, an **ActionEvent** object is generated.

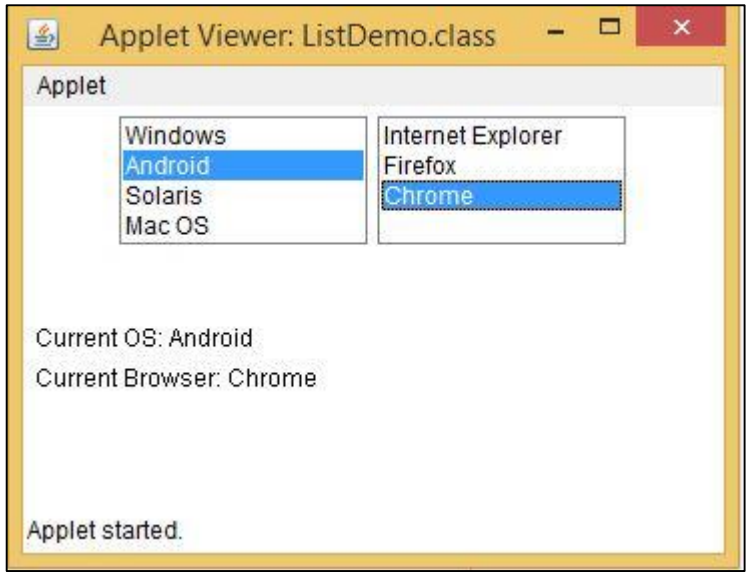
10.7.1.3. AWT Controls Fundamentals

- Its **getActionCommand()** method can be used to retrieve the name of the newly selected item. Also, each time an item is selected or deselected with a single click, an **ItemEvent** object is generated.
- Its **getStateChange()** method can be used to determine whether a selection or deselection triggered this event.
- The **getItemSelectable()** returns a reference to the object that triggered this event.
- Here is an example that converts the Choice controls in the preceding section into List components, one multiple choice and the other single choice.

10.7.1.3. AWT Controls Fundamentals

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

public class ListDemo extends Applet implements ActionListener {
    List os, browser; String msg = "";
    public void init() {
        os = new List(4, true);
        browser = new List(4, false);
        // add items to os list
        os.add("Windows"); os.add("Android");
        os.add("Solaris"); os.add("Mac OS");
        // add items to browser list
        browser.add("Internet Explorer"); browser.add("Firefox");
        browser.add("Chrome"); browser.select(1);
        // add lists to window
        add(os); add(browser);
        // register to receive action events
        os.addActionListener(this); browser.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae) {
        repaint();
    }
    // Display current selections.
    public void paint(Graphics g) {
        int idx[];
        msg = "Current OS: ";
        idx = os.getSelectedIndexes();
        for(int i=0; i<idx.length; i++)
        msg += os.getItem(idx[i]) + " ";
        g.drawString(msg, 6, 120);
        msg = "Current Browser: ";
        msg += browser.getSelectedItem();
        g.drawString(msg, 6, 140);
    }
}
```

Sample Output



10.7.1.3. AWT Controls Fundamentals

Managing Scroll Bars

- **Scroll bars** are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically.
- A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow.
- The current value of the scroll bar relative to its minimum and maximum values is indicated by the *slider box* (or *thumb*) for the scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value.

10.7.1.3. AWT Controls Fundamentals

- In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this action translates into some form of page up and page down.
- Scroll bars are encapsulated by the **Scrollbar** class. Scrollbar defines the following constructors:

Scrollbar() throws HeadlessException

Scrollbar(int *style*) throws HeadlessException

Scrollbar(int *style*, int *initialValue*, int *thumbSize*, int *min*, int *max*) throws HeadlessException

10.7.1.3. AWT Controls Fundamentals

- The first form creates a vertical scroll bar.
- The second and third forms allow you to specify the orientation of the scroll bar. If style is **Scrollbar.VERTICAL**, a vertical scroll bar is created. If style is **Scrollbar.HORIZONTAL**, the scroll bar is horizontal.
- In the third form of the constructor, the initial value of the scroll bar is passed in *initialValue*. The number of units represented by the height of the thumb is passed in *thumbSize*. The minimum and maximum values for the scroll bar are specified by *min* and *max*.
- If you construct a scroll bar by using one of the first two constructors, then you need to set its parameters by using **void setValues(int *initialValue*, int *thumbSize*, int *min*, int *max*)**, before it can be used.

10.7.1.3. AWT Controls Fundamentals

- To obtain the current value of the scroll bar, call **int getValue().** It returns the current setting.
- To set the current value, call **void setValue(int *newValue*).** Here, *newValue* specifies the new value for the scroll bar. When you set a value, the slider box inside the scroll bar will be positioned to reflect the new value.
- You can also retrieve the minimum and maximum values via **int getMinimum()** and **int getMaximum().** They return the requested quantity.
- By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line.

10.7.1.3. AWT Controls Fundamentals

- You can change this increment by calling **void setUnitIncrement(int *newIncr*).** By default, page-up and page-down increments are 10. You can change this value by calling **void setBlockIncrement(int *newIncr*).**

Handling Scroll Bars

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

10.7.1.3. AWT Controls Fundamentals

- The types of adjustment events are as follows.

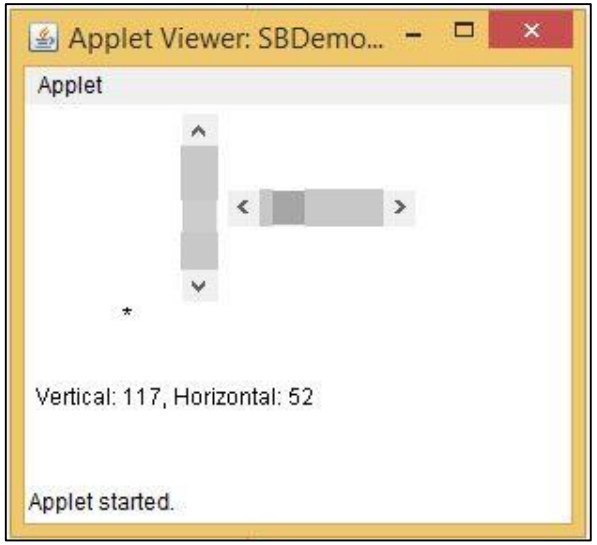
BLOCK_DECREMENT	A page-down event has been generated.
BLOCK_INCREMENT	A page-up event has been generated.
TRACK	An absolute tracking event has been generated.
UNIT_DECREMENT	The line-down button in a scroll bar has been pressed.
UNIT_INCREMENT	The line-up button in a scroll bar has been pressed.

- The following example creates both a vertical and a horizontal scroll bar. The current settings of the scroll bars are displayed.
- If you drag the mouse while inside the window, the coordinates of each drag event are used to update the scroll bars. An asterisk is displayed at the current drag position. The use of **setPreferredSize()** to set the size of the Scrollbars.

10.7.1.3. AWT Controls Fundamentals

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

public class SBDemo extends Applet
implements AdjustmentListener, MouseMotionListener {
    String msg = "";
    Scrollbar vertSB, horzSB;
    public void init() {
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));
        vertSB = new Scrollbar(Scrollbar.VERTICAL,0, 1, 0, height);
        vertSB.setPreferredSize(new Dimension(20, 100));
        horzSB = new Scrollbar(Scrollbar.HORIZONTAL,0, 1, 0, width);
        horzSB.setPreferredSize(new Dimension(100, 20));
        add(vertSB);add(horzSB);
        // register to receive adjustment events
        vertSB.addAdjustmentListener(this);
        horzSB.addAdjustmentListener(this);
        addMouseMotionListener(this);
    }

    public void adjustmentValueChanged(AdjustmentEvent ae) {
        repaint();
    }
    // Update scroll bars to reflect mouse dragging.
    public void mouseDragged(MouseEvent me) {
        int x = me.getX(), y = me.getY();
        vertSB.setValue(y); horzSB.setValue(x);
        repaint();
    }
    // Update scroll bars to reflect mouse dragging.
    public void mouseDragged(MouseEvent me) {
        int x = me.getX(), y = me.getY();
        vertSB.setValue(y); horzSB.setValue(x);
        repaint();
    }
    // Necessary for MouseMotionListener
    public void mouseMoved(MouseEvent me) {
    }
    // Display current value of scroll bars.
    public void paint(Graphics g) {
        msg = "Vertical: " + vertSB.getValue();
        msg += ", Horizontal: " + horzSB.getValue();
        g.drawString(msg, 6, 160);
        // show current mouse drag position
        g.drawString("*", horzSB.getValue(),
        vertSB.getValue());
    }
}
```

Sample Output



10.7.1.3. AWT Controls Fundamentals

Using a TextField

- The **TextField** class implements a single-line text-entry area, usually called an *edit control.* Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections.
- TextField is a subclass of **TextComponent**. TextField defines the following constructors:

TextField( ) throws HeadlessException

TextField(int *numChars*) throws HeadlessException

TextField(String *str*) throws HeadlessException

TextField(String *str*, int *numChars*) throws HeadlessException

10.7.1.3. AWT Controls Fundamentals

- The first version creates a default text field. The second form creates a text field that is *numChars* characters wide. The third form initializes the text field with the string contained in *str*. The fourth form initializes a text field and sets its width.
- TextField (and its superclass TextComponent) provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, **String getText()**. To set the text, call **void setText(String *str*)**. Here, *str* is the new string.
- The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using **void select(int *startIndex*, int *endIndex*)**.

10.7.1.3. AWT Controls Fundamentals

- Your program can obtain the currently selected text by calling **String getSelectedText()**.
- The **getSelectedText()** returns the selected text. The **select()** method selects the characters beginning at *startIndex* and ending at *endIndex* –1.
- You can control whether the contents of a text field may be modified by the user by calling **void setEditable(boolean canEdit)**. You can determine editability by calling **boolean isEditable( )**.
- The isEditable( ) returns true if the text may be changed and false if not. In setEditable( ), if *canEdit* is true, the text may be changed. If it is false, the text cannot be altered.

10.7.1.3. AWT Controls Fundamentals

- There may be times when you will want the user to enter text that is not displayed, such as a password.
- You can disable the echoing of the characters as they are typed by calling **void setEchoChar(char *ch*)**. This method specifies a single character that the TextField will display when characters are entered (thus, the actual characters typed will not be shown). Here, *ch* specifies the character to be echoed. If ch is zero, then normal echoing is restored.
- You can check a text field to see if it is in this mode with the **boolean echoCharIsSet()** method**.**
- You can retrieve the echo character by calling the **char getEchoChar()** method.

10.7.1.3. AWT Controls Fundamentals

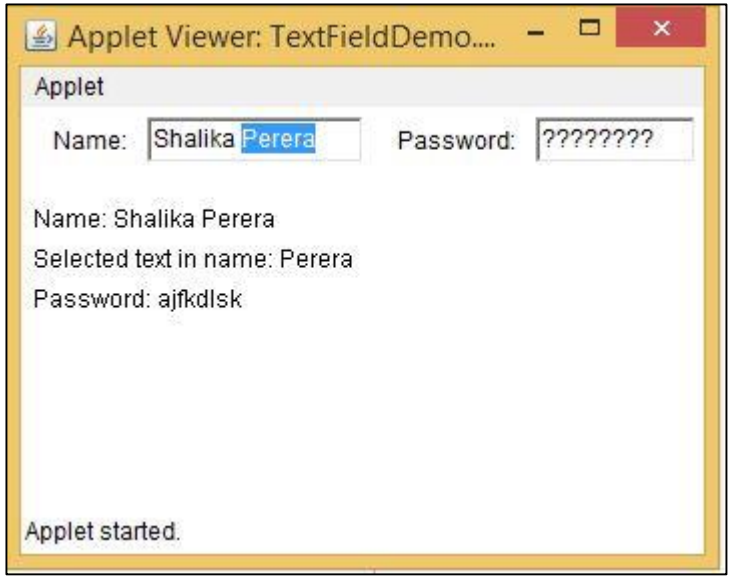
Handling a TextField

- Since text fields perform their own editing functions, your program generally will not respond to individual key events that occur within a text field.
- However, you may want to respond when the user presses enter. When this occurs, an action event is generated.
- Here is an example that creates the classic user name and password screen.

10.7.1.3. AWT Controls Fundamentals

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

public class TextFieldDemo extends Applet implements ActionListener {
    TextField name, pass;
    public void init() {
        Label namep = new Label("Name: ", Label.RIGHT);
        Label passp = new Label("Password: ", Label.RIGHT);
        name = new TextField(12);
        pass = new TextField(8);
        pass.setEchoChar('?');
        add(namep);
        add(name);
        add(passp);
        add(pass);
        // register to receive action events
        name.addActionListener(this);
        pass.addActionListener(this);
    }
    // User pressed Enter.
    public void actionPerformed(ActionEvent ae) {
        repaint();
    }
    public void paint(Graphics g) {
        g.drawString("Name: " + name.getText(), 6, 60);
        g.drawString("Selected text in name: "
        + name.getSelectedText(), 6, 80);
        g.drawString("Password: " + pass.getText(), 6, 100);
    }
}
```

Sample Output



10.7.1.3. AWT Controls Fundamentals

Using a TextArea

- Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called **TextArea**.
- Following are the constructors for TextArea.

TextArea( ) throws HeadlessException

TextArea(int *numLines, int numChars) throws HeadlessException*

TextArea(String *str) throws HeadlessException*

TextArea(String *str, int numLines, int numChars) throws HeadlessException*

TextArea(String *str, int numLines, int numChars, int sBars) throws HeadlessException*

10.7.1.3. AWT Controls Fundamentals

- Here, *numLines* specifies the height, in lines, of the text area, and *numChars* specifies its width, in characters.
- Initial text can be specified by *str*.
- In the fifth form, you can specify the scroll bars that you want the control to have. *sBars* must be one of these values:

SCROLLBARS_BOTH	SCROLLBARS_NONE
SCROLLBARS_HORIZONTAL_ONLY	SCROLLBARS_VERTICAL_ONLY

10.7.1.3. AWT Controls Fundamentals

- TextArea is a subclass of **TextComponent**. Therefore, it supports the getText(), setText(), getSelectedText(), select(), isEditable(), and setEditable() methods previously described.
- TextArea adds the following editing methods:

  void append(String *str)

  void insert(String *str, int index)

  void replaceRange(String *str, int startIndex, int endIndex)

- The **append()** method appends the string specified by *str* to the end of the current text. The **insert()** inserts the string passed in *str* at the specified index.

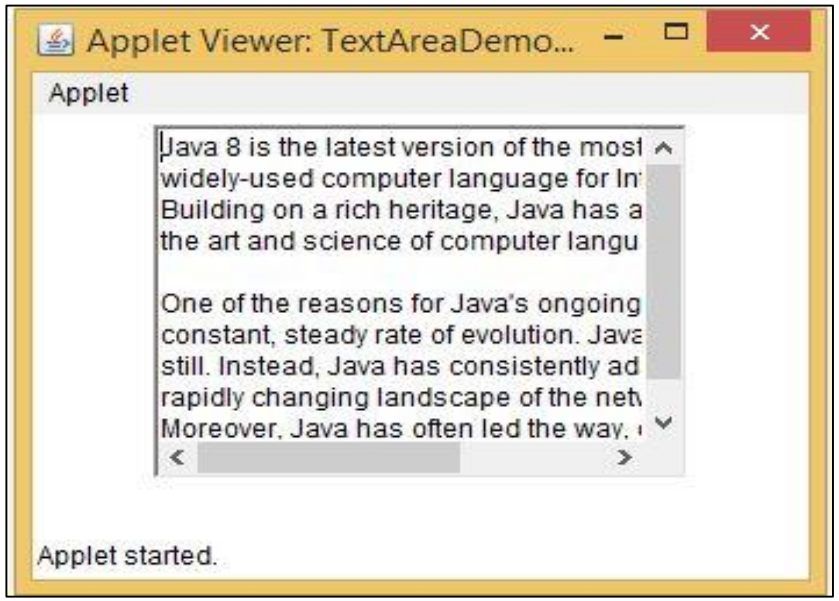
10.7.1.3. AWT Controls Fundamentals

- To replace text, call **replaceRange().** It replaces the characters from *startIndex* to *endIndex–1*, with the replacement text passed in *str*.
- Text areas are almost self-contained controls. Your program incurs virtually no management overhead.
- Normally, your program simply obtains the current text when it is needed. You can, however, listen for **TextEvents**, if you choose.
- The following program creates a TextArea control.

10.7.1.3. AWT Controls Fundamentals

```
import java.awt.*;
import java.applet.*;

public class TextAreaDemo extends Applet {
    public void init() {
        String val =
        "Java 8 is the latest version of the most\n" +
        "widely-used computer language for Internet programming.\n" +
        "Building on a rich heritage, Java has advanced both\n" +
        "the art and science of computer language design.\n\n" +
        "One of the reasons for Java's ongoing success is its\n" +
        "constant, steady rate of evolution. Java has never stood\n" +
        "still. Instead, Java has consistently adapted to the\n" +
        "rapidly changing landscape of the networked world.\n" +
        "Moreover, Java has often led the way, charting the\n" +
        "course for others to follow.";
        TextArea text = new TextArea(val, 10, 30);
        add(text);
    }
}
```

Sample Output



10.7.1.4. Layout Managers

- All of the components that we have shown so far have been positioned by the default layout manager. A layout manager automatically arranges your controls within a window by using some type of algorithm.
- A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout()** method.
- If no call to setLayout() is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

10.7.1.4. Layout Managers

- The setLayout() method has the following general form:

  void setLayout(LayoutManager *layoutObj*)

- Here, *layoutObj* is a reference to the desired layout manager.
- If you wish to disable the layout manager and position components manually, pass **null** for *layoutObj*.
- If you do this, you will need to determine the shape and position of each component manually, using the **setBounds()** method defined by Component. Normally, you will want to use a layout manager.
- Each layout manager keeps track of a list of components that are stored by their names.

10.7.1.4. Layout Managers

- The layout manager is notified each time you add a component to a container. Whenever the container needs to be resized, the layout manager is consulted via its **minimumLayoutSize()** and **preferredLayoutSize()** methods.
- Each component that is being managed by a layout manager contains the **getPreferredSize()** and **getMinimumSize()** methods. These return the preferred and minimum size required to display each component.
- The layout manager will honor these requests if at all possible, while maintaining the integrity of the layout policy.

10.7.1.4. Layout Managers

- You may override these methods for controls that you subclass. Default values are provided otherwise.
- Java has several predefined **LayoutManager** classes and you can use the layout manager that best fits your application.

FlowLayout

- **FlowLayout** is the default layout manager. This is the layout manager that the preceding examples have used. FlowLayout implements a simple layout style, which is similar to how words flow in a text editor.
- The direction of the layout is governed by the container's component orientation property, which, by default, is left to right, top to bottom.

10.7.1.4. Layout Managers

- Therefore, by default, components are laid out line-by-line beginning at the upper-left corner. In all cases, when a line is filled, layout advances to the next line. A small space is left between each component, above and below, as well as left and right.
- Here are the constructors for FlowLayout:

  FlowLayout( )

  FlowLayout(int *how*)

  FlowLayout(int *how*, int *horz*, int *vert*)

- The first form creates the default layout, which centers components and leaves five pixels of space between each component. The second form lets you specify how each line is aligned.

10.7.1.4. Layout Managers

- Valid values for *how* are FlowLayout.LEFT, FlowLayout.CENTER, FlowLayout.RIGHT, FlowLayout.LEADING and FlowLayout.TRAILING. These values specify left, center, right, leading edge, and trailing edge alignment, respectively.
- The third constructor allows you to specify the horizontal and vertical space left between components in *horz* and *vert* respectively.
- Here is a version of the CheckboxDemo applet shown earlier in this chapter and modified. So that it uses left-aligned flow layout:

10.7.1.4. Layout Managers

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

public class FlowLayoutDemo extends Applet implements ItemListener {
    String msg = "";
    Checkbox windows, android, solaris, mac;
    public void init() {
        // set left-aligned flow layout
        setLayout(new FlowLayout(FlowLayout.LEFT));
        windows = new Checkbox("Windows", null, true);
        android = new Checkbox("Android");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");
        add(windows); add(android);
        add(solaris);add(mac);
        // register to receive item events
        windows.addItemListener(this);
        android.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
    }

    // Repaint when status of a check box changes.
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }

    // Display current state of the check boxes.
    public void paint(Graphics g) {
        msg = "Current state: ";
        g.drawString(msg, 6, 80);
        msg = " Windows: " + windows.getState();
        g.drawString(msg, 6, 100);
        msg = " Android: " + android.getState();
        g.drawString(msg, 6, 120);
        msg = " Solaris: " + solaris.getState();
        g.drawString(msg, 6, 140);
        msg = " Mac: " + mac.getState();
        g.drawString(msg, 6, 160);
    }
}
```

Sample Output

10.7.1.4. Layout Managers

BorderLayout

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

BorderLayout( )

BorderLayout(int *horz*, int *vert*)

10.7.1.4. Layout Managers

- The first form creates a default border layout. The second allows you to specify the horizontal and vertical space left between components in *horz* and *vert* respectively.
- BorderLayout defines BorderLayout.CENTER, BorderLayout.EAST, BorderLayout.NORTH, BorderLayout.SOUTH and BorderLayout.WEST constants that specify the regions.
- When adding components, you will use these constants with the following form of **add()**, which is defined by Container:

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

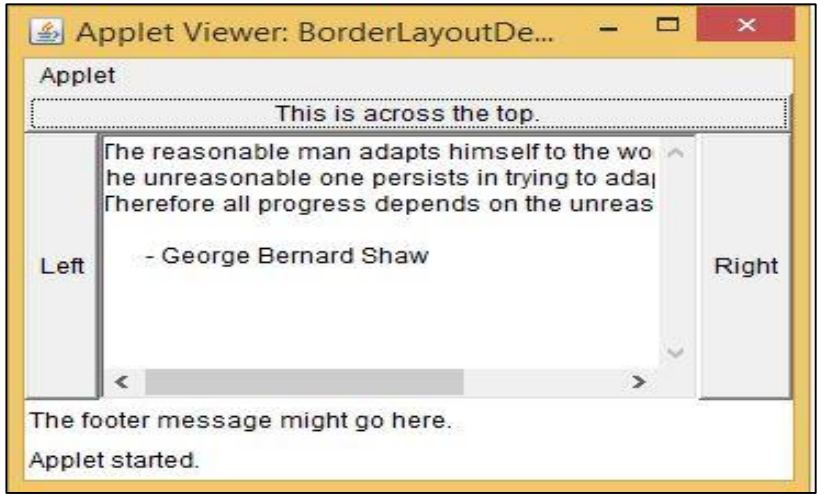
- Here, *compRef* is a reference to the component to be added and region specifies where the component will be added.

10.7.1.4. Layout Managers

- Here is an example of a BorderLayout with a component in each layout area:

```
import java.awt.*;
import java.applet.*;

public class BorderLayoutDemo extends Applet {
    public void init() {
        setLayout(new BorderLayout());
        add(new Button("This is across the top."),
        BorderLayout.NORTH);
        add(new Label("The footer message might go here."),
        BorderLayout.SOUTH);
        add(new Button("Right"), BorderLayout.EAST);
        add(new Button("Left"), BorderLayout.WEST);
        String msg = "The reasonable man adapts " +
        "himself to the world;\n" +
        "the unreasonable one persists in " +
        "trying to adapt the world to himself.\n" +
        "Therefore all progress depends " +
        "on the unreasonable man.\n\n" +
        " - George Bernard Shaw\n\n";
        add(new TextArea(msg), BorderLayout.CENTER);
    }
}
```

Sample Output



10.7.1.4. Layout Managers

Using Insets

- Sometimes you will want to leave a small amount of space between the container that holds your components and the window that contains it. To do this, override the **getInsets()** method that is defined by Container.
- This method returns an **Insets** object that contains the top, bottom, left, and right inset to be used when the container is displayed. These values are used by the layout manager to inset the components when it lays out the window.
- The constructor for Insets is shown here:

Insets(int *top*, int *left*, int *bottom*, int *right*)

10.7.1.4. Layout Managers

- The values passed in *top*, *left*, *bottom*, and *right* specify the amount of space between the container and its enclosing window.
- The **getInsets()** method has this general form:

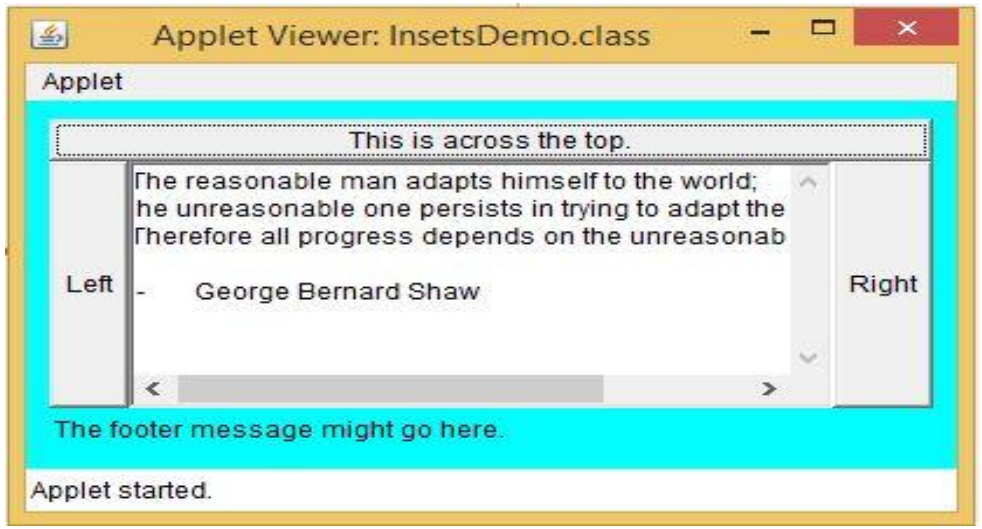
  Insets getInsets()

- When overriding this method, you must return a new Insets object that contains the inset spacing you desire.
- Here is the preceding BorderLayout example modified so that it insets its components ten pixels from each border. The background color has been set to cyan to help make the insets more visible.

10.7.1.4. Layout Managers

```
import java.awt.*;
import java.applet.*;

public class InsetsDemo extends Applet {
    public void init() {
        // set background color so insets can be easily seen
        setBackground(Color.cyan);
        setLayout(new BorderLayout());
        add(new Button("This is across the top."),
        BorderLayout.NORTH);
        add(new Label("The footer message might go here."),
        BorderLayout.SOUTH);
        add(new Button("Right"), BorderLayout.EAST);
        add(new Button("Left"), BorderLayout.WEST);
        String msg = "The reasonable man adapts " +
        "himself to the world;\n" +
        "the unreasonable one persists in " +
        "trying to adapt the world to himself.\n" +
        "Therefore all progress depends " +
        "on the unreasonable man.\n\n" +
        " - George Bernard Shaw\n\n";
        add(new TextArea(msg), BorderLayout.CENTER);
    }
    // add insets
    public Insets getInsets() {
        return new Insets(10, 10, 10, 10);
    }
}
```

Sample Output



10.7.1.4. Layout Managers

GridLayout

- **GridLayout** lays out components in a two-dimensional grid. When you instantiate a GridLayout, you define the number of rows and columns.
- The constructors supported by GridLayout are shown here:

  GridLayout( )

  GridLayout(int *numRows*, int *numColumns*)

  GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*)

- The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns.

10.7.1.4. Layout Managers

- The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert* respectively. Either *numRows* or *numColumns* can be zero.

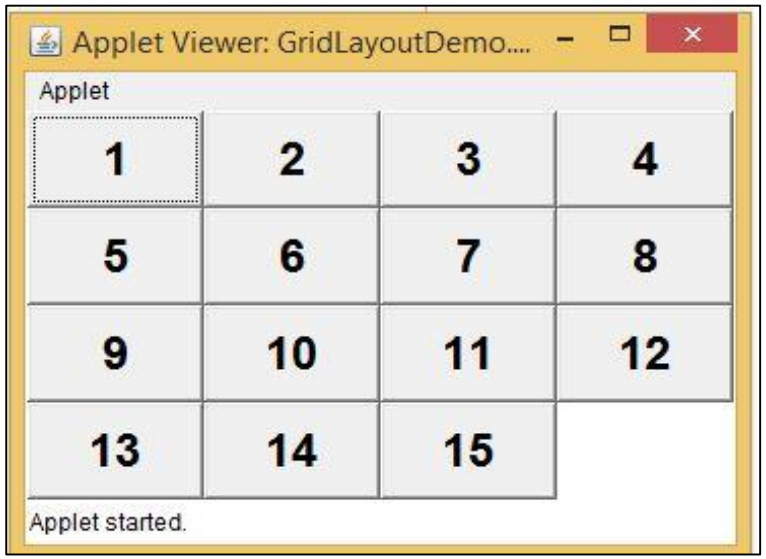
- Specifying *numRows* as zero allows for unlimited-length columns. Specifying *numColumns* as zero allows for unlimited-length rows.

- Here is a sample program that creates a 4×4 grid and fills it in with 15 buttons, each labeled with its index:

10.7.1.4. Layout Managers

```
import java.awt.*;
import java.applet.*;

public class GridLayoutDemo extends Applet {
    static final int n = 4;
    public void init() {
        setLayout(new GridLayout(n, n));
        setFont(new Font("SansSerif", Font.BOLD, 24));
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                int k = i * n + j;
                if(k > 0)
                add(new Button("" + k));
            }
        }
    }
}
```

Sample Output



10.7.1.4. Layout Managers

CardLayout

- The **CardLayout** class is unique among the other layout managers in that it stores several different layouts.
- Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time.
- This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input.
- You can prepare the other layouts and have them hidden, ready to be activated when needed.

10.7.1.4. Layout Managers

- CardLayout provides these two constructors:

  CardLayout( )

  CardLayout(int *horz*, int *vert*)

- The first form creates a default card layout. The second form allows you to specify the horizontal and vertical space left between components in *horz* and *vert* respectively.
- Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type **Panel**. This panel must have CardLayout selected as its layout manager.

10.7.1.4. Layout Managers

- The cards that form the deck are also typically objects of type Panel. Thus, you must create a panel that contains the deck and a panel for each card in the deck.
- Next, you add to the appropriate panel the components that form each card. You then add these panels to the panel for which CardLayout is the layout manager.
- Finally, you add this panel to the window. Once these steps are complete, you must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck. When card panels are added to a panel, they are usually given a name.

10.7.1.4. Layout Managers

- Thus, most of the time, you will use this form of **void add(Component *panelRef*, Object *name*)** when adding cards to a panel. Here, *name* is a string that specifies the name of the card whose panel is specified by *panelRef*.
- After you have created a deck, your program activates a card by calling one of the following methods defined by CardLayout:

void first(Container *deck*)

void last(Container *deck*)

void next(Container *deck*)

void previous(Container *deck*)

void show(Container *deck*, String *cardName*)

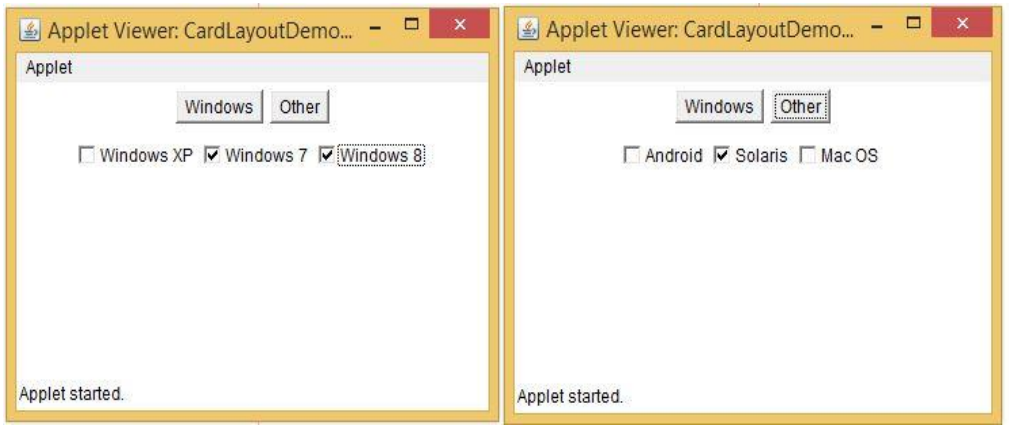
10.7.1.4. Layout Managers

- Here, *deck* is a reference to the container (usually a panel) that holds the cards, and *cardName* is the name of a card.
- Calling **first()** causes the first card in the deck to be shown. To show the last card, call **last()**. To show the next card, call **next()**. To show the previous card, call **previous()**. Both **next()** and **previous()** automatically cycle back to the top or bottom of the deck, respectively. The **show()** method displays the card whose name is passed in *cardName*.
- The following example creates a two-level card deck that allows the user to select an operating system. Windows-based operating systems are displayed in one card. Mac OS and Solaris are displayed in the other card.

10.7.1.4. Layout Managers

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class CardLayoutDemo extends Applet
implements ActionListener, MouseListener {
    Checkbox windowsXP, windows7, windows8, android, solaris, mac;
    Panel osCards;
    CardLayout cardLO;
    Button Win, Other;
    public void init() {
        Win = new Button("Windows");
        Other = new Button("Other");
        add(Win); add(Other);
        cardLO = new CardLayout();
        osCards = new Panel();
        osCards.setLayout(cardLO); // set panel layout to card layout
        windowsXP = new Checkbox("Windows XP", null, true);
        windows7 = new Checkbox("Windows 7", null, false);
        windows8 = new Checkbox("Windows 8", null, false);
        android = new Checkbox("Android"); solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");
        // add Windows check boxes to a panel
        Panel winPan = new Panel();
        winPan.add(windowsXP); winPan.add(windows7); winPan.add(windows8);
        // Add other OS check boxes to a panel
        Panel otherPan = new Panel();
        otherPan.add(android); otherPan.add(solaris); otherPan.add(mac);
        // add panels to card deck panel
        osCards.add(winPan, "Windows"); osCards.add(otherPan, "Other");
        // add cards to main applet panel
        add(osCards);
        // register to receive action events
        Win.addActionListener(this); Other.addActionListener(this);
        // register mouse events
        addMouseListener(this);
    }
    // Cycle through panels.
    public void mousePressed(MouseEvent me) {
        cardLO.next(osCards);
    }
    // Provide empty implementations for the other MouseListener methods.
    public void mouseClicked(MouseEvent me) {
    }
    public void mouseEntered(MouseEvent me) {
    }
    public void mouseExited(MouseEvent me) {
    }
    public void mouseReleased(MouseEvent me) {
    }
    public void actionPerformed(ActionEvent ae) {
        if(ae.getSource() == Win) {
            cardLO.show(osCards, "Windows");
        }
        else {
            cardLO.show(osCards, "Other");
        }
    }
}
```

Sample Output



10.7.1.4. Layout Managers

- In the sample output, each card is activated by pushing its button. You can also cycle through the cards by clicking the mouse.

GridBagLayout

- Although the preceding layouts are perfectly acceptable for many uses, some situations will require that you take a bit more control over how the components are arranged. A good way to do this is to use a grid bag layout, which is specified by the **GridBagLayout** class.
- What makes the grid bag useful is that you can specify the relative placement of components by specifying their positions within cells inside a grid. The key to the grid bag is that each component can be a different size, and each row in the grid can have a different number of columns.

10.7.1.4. Layout Managers

- This is why the layout is called a *grid bag*. It's a collection of small grids joined together.
- The location and size of each component in a grid bag are determined by a set of constraints linked to it. The constraints are contained in an object of type **GridBagConstraints**. Constraints include the height and width of a cell, and the placement of a component, its alignment, and its anchor point within the cell.
- The general procedure for using a grid bag is to first create a new GridBagLayout object and to make it the current layout manager. Then, set the constraints that apply to each component that will be added to the grid bag.

10.7.1.4. Layout Managers

- Finally, add the components to the layout manager. Although GridBagLayout is a bit more complicated than the other layout managers, it is still quite easy to use once you understand how it works.
- GridBagLayout defines only one constructor as **GridBagLayout()**.
- GridBagLayout defines several methods, of which many are protected and not for general use. There is one method, however, that you must use **void setConstraints(Component *comp*, GridBagConstraints *cons*).** Here, *comp* is the component for which the constraints specified by *cons* apply.
- This method sets the constraints that apply to each component in the grid bag.

10.7.1.4. Layout Managers

- The key to successfully using GridBagLayout is the proper setting of the constraints, which are stored in a GridBagConstraints object. GridBagConstraints defines several fields that you can set to govern the size, placement, and spacing of a component.
- Several constraint fields defined by GridBagConstraints are as follows.

Field	Purpose
int anchor	Specifies the location of a component within a cell. The default is GridBagConstraints.CENTER.
int fill	Specifies how a component is resized if the component is smaller than its cell. Valid values are GridBagConstraints.NONE (the default), GridBagConstraints.HORIZONTAL, GridBagConstraints.VERTICAL, GridBagConstraints.BOTH.

10.7.1.4. Layout Managers

Field	Purpose
int gridheight	Specifies the height of component in terms of cells. The default is 1.
int gridwidth	Specifies the width of component in terms of cells. The default is 1.
int gridx	Specifies the X coordinate of the cell to which the component will be added. The default value is GridBagConstraints.RELATIVE.
int gridy	Specifies the Y coordinate of the cell to which the component will be added. The default value is GridBagConstraints.RELATIVE.
Insets insets	Specifies the insets. Default insets are all zero.
int ipadx	Specifies extra horizontal space that surrounds a component within a cell. The default is 0.
int ipady	Specifies extra vertical space that surrounds a component within a cell. The default is 0.

10.7.1.4. Layout Managers

- GridBagConstraints also defines several static fields that contain standard constraint values, such as GridBagConstraints.CENTER and GridBagConstraints.VERTICAL.
- When a component is smaller than its cell, you can use the anchor field to specify where within the cell the component's top-left corner will be located.
- There are three types of values that you can give to anchor. The first are absolute:

GridBagConstraints.CENTER	GridBagConstraints.SOUTH
GridBagConstraints.EAST	GridBagConstraints.SOUTHEAST
GridBagConstraints.NORTH	GridBagConstraints.SOUTHWEST
GridBagConstraints.NORTHEAST	GridBagConstraints.WEST
GridBagConstraints.NORTHWEST	

- As their names imply, these values cause the component to be placed at the specific locations.

10.7.1.4. Layout Managers

- The second type of values that can be given to **anchor** is relative, which means the values are relative to the container's orientation, which might differ for non-Western languages.
- The relative values are shown here:

GridBagConstraints.FIRST_LINE_END	GridBagConstraints.LINE_END
GridBagConstraints.FIRST_LINE_START	GridBagConstraints.LINE_START
GridBagConstraints.LAST_LINE_END	GridBagConstraints.PAGE_END
GridBagConstraints.LAST_LINE_START	GridBagConstraints.PAGE_START

- Their names describe the placement.

10.7.1.4. Layout Managers

- The third type of values that can be given to anchor allows you to position components relative to the baseline of the row. These values are shown here:

GridBagConstraints.BASELINE	GridBagConstraints.BASELINE_LEADING
GridBagConstraints.BASELINE_TRAILING	GridBagConstraints.ABOVE_BASELINE
GridBagConstraints.ABOVE_BASELINE_LEADING	GridBagConstraints.ABOVE_BASELINE_TRAILING
GridBagConstraints.BELOW_BASELINE	GridBagConstraints.BELOW_BASELINE_LEADING
GridBagConstraints. BELOW_BASELINE_TRAILING	

- The horizontal position can be either centered, against the leading edge (LEADING), or against the trailing edge (TRAILING).

10.7.1.4. Layout Managers

- In general, the values of **weightx** and **weighty** fields determine how much of the extra space within a container is allocated to each row and column.
- By default, both these values are zero. When all values within a row or a column are zero, extra space is distributed evenly between the edges of the window.
- By increasing the weight, you increase that row or column's allocation of space proportional to the other rows or columns. The best way to understand how these values work is to experiment with them a bit.
- The **gridwidth** variable lets you specify the width of a cell in terms of cell units. The default is 1.

10.7.1.4. Layout Managers

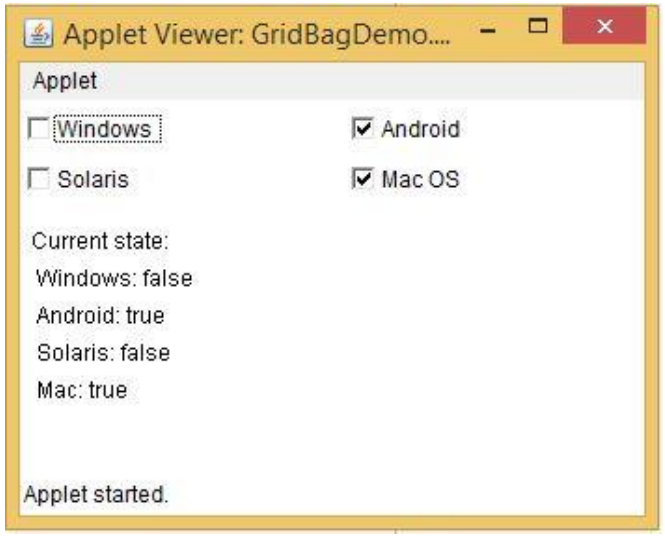
- To specify that a component use the remaining space in a row, use **GridBagConstraints.REMAINDER**. To specify that a component use the next-to-last cell in a row, use **GridBagConstraints.RELATIVE**. The gridheight constraint works the same way, but in the vertical direction.
- You can specify a padding value that will be used to increase the minimum size of a cell. To pad horizontally, assign a value to *ipadx*. To pad vertically, assign a value to *ipady*.
- Here is an example that uses GridBagLayout to demonstrate several of the points just discussed.

10.7.1.4. Layout Managers

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class GridBagDemo extends Applet implements ItemListener {
    String msg = "";
    Checkbox windows, android, solaris, mac;
    public void init() {
        GridBagLayout gbag = new GridBagLayout();
        GridBagConstraints gbc = new GridBagConstraints();
        setLayout(gbag);
        // Define check boxes.
        windows = new Checkbox("Windows ", null, true);
        android = new Checkbox("Android");solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");
        // Define the grid bag.
        // Use default row weight of 0 for first row.
        gbc.weightx = 1.0; // use a column weight of 1
        gbc.ipadx = 200; // pad by 200 units
        gbc.insets = new Insets(4, 4, 0, 0); // inset slightly from top left
        gbc.anchor = GridBagConstraints.NORTHEAST;
        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(windows, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(android, gbc);
        // Give second row a weight of 1.
        gbc.weighty = 1.0;
        gbc.gridwidth = GridBagConstraints.RELATIVE;
        gbag.setConstraints(solaris, gbc);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbag.setConstraints(mac, gbc);
        // Add the components.
        add(windows); add(android); add(solaris); add(mac);
        // Register to receive item events.
        windows.addItemListener(this);
        android.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
    }

    // Repaint when status of a check box changes.
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }
    // Display current state of the check boxes.
    public void paint(Graphics g) {
        msg = "Current state: ";
        g.drawString(msg, 6, 80);
        msg = " Windows: " + windows.getState();
        g.drawString(msg, 6, 100);
        msg = " Android: " + android.getState();
        g.drawString(msg, 6, 120);
        msg = " Solaris: " + solaris.getState();
        g.drawString(msg, 6, 140);
        msg = " Mac: " + mac.getState();
        g.drawString(msg, 6, 160);
    }
}
```

10.7.1.4. Layout Managers

- Sample output produced by the program is shown here.



- In this layout, the operating system check boxes are positioned in a 2×2 grid. Each cell has a horizontal padding of 200.
- Each component is inset slightly (by 4 units) from the top left. The column weight is set to 1, which causes any extra horizontal space to be distributed evenly between the columns.
- The first row uses a default weight of 0; the second has a weight of 1. This means that any extra vertical space is added to the second row.

10.7.1.5 Menu bars and Menus

- A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This concept is implemented in the AWT by the three classes: MenuBar, Menu, and MenuItem.
- In general, a **Menu bar** contains one or more **Menu** objects. Each Menu object contains a list of **MenuItem** objects. Each MenuItem object represents something that can be selected by the user.
- Since Menu is a subclass of MenuItem, a hierarchy of nested submenus can be created. It is also possible to include checkable menu items. These are menu options of type **CheckboxMenuItem** and will have a check mark next to them when they are selected. The first form creates an empty menu.

10.7.1.5 Menu bars and Menus

- To create a menu bar, first create an instance of **MenuBar**. This class defines only the default constructor. Next, create instances of Menu that will define the selections displayed on the bar.
- Following are the constructors for Menu:

  Menu() throws HeadlessException

  Menu(String *optionName*) throws HeadlessException

  Menu(String *optionName*, boolean *removable*) throws HeadlessException

- Here, *optionName* specifies the name of the menu selection. If *removable* is true, the menu can be removed and allowed to float free. Otherwise, it will remain attached to the menu bar.(Removable menus are implementation-dependent.)

10.7.1.5 Menu bars and Menus

- Individual menu items are of type **MenuItem**. It defines these constructors:

  MenuItem() throws HeadlessException

  MenuItem(String *itemName*) throws HeadlessException

  MenuItem(String *itemName*, MenuShortcut *keyAccel*) throws HeadlessException

- Here, *itemName* is the name shown in the menu, and *keyAccel* is the menu shortcut for this item.
- You can disable or enable a menu item by using the **void setEnabled(boolean *enabledFlag*)** method.
- If the argument *enabledFlag* is true, the menu item is enabled. If false, the menu item is disabled.

10.7.1.5 Menu bars and Menus

- You can determine an item's status by calling **boolean isEnabled()**. This method returns true if the menu item on which it is called is enabled. Otherwise, it returns false.
- You can change the name of a menu item by calling **void setLabel(String newName)**. You can retrieve the current name by using **String getLabel()**.
- Here, *newName* becomes the new name of the invoking menu item. The getLabel() returns the current name.
- You can create a checkable menu item by using a subclass of MenuItem called **CheckboxMenuItem**.

10.7.1.5 Menu bars and Menus

- It has three constructors:

  CheckboxMenuItem() throws HeadlessException

  CheckboxMenuItem(String *itemName*) throws HeadlessException

  CheckboxMenuItem(String *itemName*, boolean *on*) throws HeadlessException

- Here, *itemName* is the name shown in the menu. Checkable items operate as toggles. Each time one is selected, its state changes.
- In the first two forms, the checkable entry is unchecked. In the third form, if *on* is true, the checkable entry is initially checked. Otherwise, it is cleared.

10.7.1.5 Menu bars and Menus

- You can obtain the status of a checkable item by calling **boolean getState()**. You can set it to a known state by using **void setState(boolean *checked*)**.
- If the item is checked, getState() returns true. Otherwise, it returns false. To check an item, pass true to setState(). To clear an item, pass false.
- Once you have created a menu item, you must add the item to a Menu object by using **MenuItem add(MenuItem item)**. Here, *item* is the item being added. Items are added to a menu in the order in which the calls to add() take place. The *item* is returned.

10.7.1.5 Menu bars and Menus

- Once you have added all items to a Menu object, you can add that object to the menu bar by using this version of **Menu add(Menu *menu*)** defined by MenuBar. Here, *menu* is the menu being added. The *menu* is returned.
- Menus generate events only when an item of type MenuItem or CheckboxMenuItem is selected. For example, they do not generate events when a menu bar is accessed to display a drop-down menu. Each time a menu item is selected, an **ActionEvent** object is generated.
- By default, the action command string is the name of the menu item. However, you can specify a different action command string by calling **setActionCommand()** on the menu item.

10.7.1.5 Menu bars and Menus

- Each time a check box menu item is checked or unchecked, an **ItemEvent** object is generated. Thus, you must implement the **ActionListener** and/or **ItemListener** interfaces in order to handle these menu events.
- The **Object getItem()** method of ItemEvent returns a reference to the item that generated this event.
- Following is an example that adds a series of nested menus to a pop-up window. The item selected is displayed in the window. The state of the two check box menu items is also displayed.

10.7.1.5 Menu bars and Menus

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

// Create a subclass of Frame.
class MenuFrame extends Frame {
    String msg = "";
    CheckboxMenuItem debug, test;
    MenuFrame(String title) {
        super(title);
        // create menu bar and add it to frame
        MenuBar mbar = new MenuBar();
        setMenuBar(mbar);
        // create the menu items
        Menu file = new Menu("File");
        MenuItem item1, item2, item3, item4, item5;
        file.add(item1 = new MenuItem("New..."));
        file.add(item2 = new MenuItem("Open..."));
        file.add(item3 = new MenuItem("Close"));
        file.add(item4 = new MenuItem("-"));
        file.add(item5 = new MenuItem("Quit..."));
        mbar.add(file);
        Menu edit = new Menu("Edit");
        MenuItem item6, item7, item8, item9;
        edit.add(item6 = new MenuItem("Cut"));
        edit.add(item7 = new MenuItem("Copy"));
        edit.add(item8 = new MenuItem("Paste"));
        edit.add(item9 = new MenuItem("-"));
        Menu sub = new Menu("Special");
        MenuItem item10, item11, item12;
        sub.add(item10 = new MenuItem("First"));
        sub.add(item11 = new MenuItem("Second"));
        sub.add(item12 = new MenuItem("Third"));
        edit.add(sub);
        // these are checkable menu items
        debug = new CheckboxMenuItem("Debug");
        edit.add(debug);
        test = new CheckboxMenuItem("Testing");
        edit.add(test); mbar.add(edit);
        // create an object to handle action and item events
        MyMenuHandler handler = new MyMenuHandler(this);
        // register it to receive those events
        item1.addActionListener(handler);
        item2.addActionListener(handler);
        item3.addActionListener(handler);
        item4.addActionListener(handler);
        item5.addActionListener(handler);
        item6.addActionListener(handler);
        item7.addActionListener(handler);
        item8.addActionListener(handler);
        item9.addActionListener(handler);
        item10.addActionListener(handler);
        item11.addActionListener(handler);
        item12.addActionListener(handler);
        debug.addItemListener(handler); test.addItemListener(handler);
        // create an object to handle window events
        MyWindowAdapter adapter = new MyWindowAdapter(this);
        // register it to receive those events
        addWindowListener(adapter);
    }
```

10.7.1.5 Menu bars and Menus

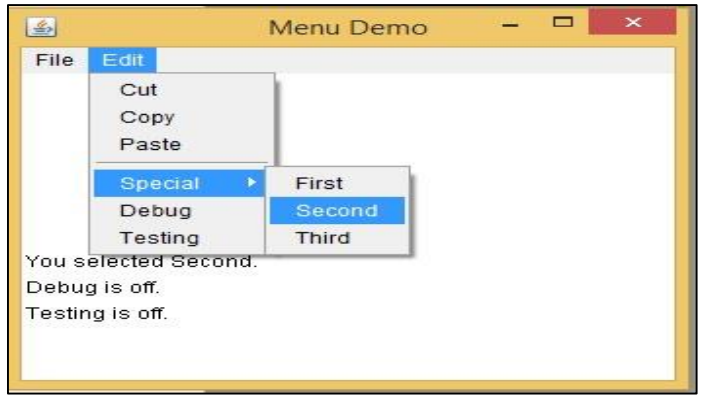
```
    public void paint(Graphics g) {
        g.drawString(msg, 10, 200);
        if(debug.getState())
            g.drawString("Debug is on.", 10, 220);
        else
            g.drawString("Debug is off.", 10, 220);
        if(test.getState())
            g.drawString("Testing is on.", 10, 240);
        else
            g.drawString("Testing is off.", 10, 240);
    }
}

class MyWindowAdapter extends WindowAdapter {
    MenuFrame menuFrame;
    public MyWindowAdapter(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
    public void windowClosing(WindowEvent we) {
        menuFrame.setVisible(false);
    }
}

class MyMenuHandler implements ActionListener, ItemListener {
    MenuFrame menuFrame;
    public MyMenuHandler(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
```

```
    // Handle action events.
    public void actionPerformed(ActionEvent ae) {
        String msg = "You selected ", arg = ae.getActionCommand();
        if(arg.equals("New..."))
            msg += "New.";
        else if(arg.equals("Open..."))
            msg += "Open.";
        else if(arg.equals("Close"))
            msg += "Close.";
        else if(arg.equals("Quit..."))
            msg += "Quit.";
        else if(arg.equals("Edit"))
            msg += "Edit.";
        else if(arg.equals("Cut"))
            msg += "Cut.";
        else if(arg.equals("Copy"))
            msg += "Copy.";
        else if(arg.equals("Paste"))
            msg += "Paste.";
        else if(arg.equals("First"))
            msg += "First.";
        else if(arg.equals("Second"))
            msg += "Second.";
        else if(arg.equals("Third"))
            msg += "Third.";
        else if(arg.equals("Debug"))
            msg += "Debug.";
        else if(arg.equals("Testing"))
            msg += "Testing.";
        menuFrame.msg = msg; menuFrame.repaint();
    }
    // Handle item events.
    public void itemStateChanged(ItemEvent ie) {
        menuFrame.repaint();
    }
```

10.7.1.5 Menu bars and Menus

```
// Create frame window.
public class MenuDemo extends Applet {
    Frame f;
    public void init() {
        f = new MenuFrame("Menu Demo");
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));
        setSize(new Dimension(width, height));
        f.setSize(width, height);
        f.setVisible(true);
    }
    public void start() {
        f.setVisible(true);
    }
    public void stop() {
        f.setVisible(false);
    }
}
```

Sample Output



- There is one other menu-related class **PopupMenu**. It works just like Menu, but produces a menu that can be displayed at a specific location.
- PopupMenu provides a flexible, useful alternative for some types of menuing situations.

10.7.1.6. Dialog boxes

- Dialog boxes are primarily used to obtain user input and are often child windows of a top-level window.
- Dialog boxes don't have menu bars, but in other respects, they function like frame windows. (You can add controls to them. For example, in the same way that you add controls to a frame window.)
- Dialog boxes may be modal or modeless.
- When a *modal* dialog box is active, all input is directed to it until it is closed. This means that you cannot access other parts of your program until you have closed the dialog box.

10.7.1.6. Dialog boxes

- When a *modeless* dialog box is active, input focus can be directed to another window in your program. Thus, other parts of your program remain active and accessible.
- In the AWT, dialog boxes are of type Dialog. Two commonly used constructors are shown here:

  Dialog(Frame *parentWindow*, boolean *mode*)

  Dialog(Frame *parentWindow*, String *title*, boolean *mode*)

- Here, *parentWindow* is the owner of the dialog box. If *mode* is true, the dialog box is modal. Otherwise, it is modeless. The title of the dialog box can be passed in *title*.

10.7.1.6. Dialog boxes

- Generally, you will subclass Dialog, adding the functionality required by your application.
- Following is a modified version of the preceding menu program that displays a modeless dialog box when the New option is chosen. Notice that when the dialog box is closed, **dispose()** is called.
- This method is defined by Window and it frees all system resources associated with the dialog box window.

10.7.1.6. Dialog boxes

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

// Create a subclass of Dialog.
class SampleDialog extends Dialog implements ActionListener {
    SampleDialog(Frame parent, String title) {
        super(parent, title, false);
        setLayout(new FlowLayout());
        setSize(300, 200);
        add(new Label("Press this button:"));
        Button b;
        add(b = new Button("Cancel"));
        b.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae) {
        dispose();
    }
    public void paint(Graphics g) {
        g.drawString("This is in the dialog box", 10, 70);
    }
}

// Create a subclass of Frame.
class MenuFrame extends Frame {
    String msg = "";
    CheckboxMenuItem debug, test;
    MenuFrame(String title) {
        super(title);
        // create menu bar and add it to frame
        MenuBar mbar = new MenuBar();
        setMenuBar(mbar);
```

```
// create the menu items
Menu file = new Menu("File");
MenuItem item1, item2, item3, item4;
file.add(item1 = new MenuItem("New..."));
file.add(item2 = new MenuItem("Open..."));
file.add(item3 = new MenuItem("Close"));
file.add(new MenuItem("-"));
file.add(item4 = new MenuItem("Quit..."));
mbar.add(file);
Menu edit = new Menu("Edit");
MenuItem item5, item6, item7;
edit.add(item5 = new MenuItem("Cut"));
edit.add(item6 = new MenuItem("Copy"));
edit.add(item7 = new MenuItem("Paste"));
edit.add(new MenuItem("-"));
Menu sub = new Menu("Special", true);
MenuItem item8, item9, item10;
sub.add(item8 = new MenuItem("First"));
sub.add(item9 = new MenuItem("Second"));
sub.add(item10 = new MenuItem("Third"));
edit.add(sub);
// these are checkable menu items
debug = new CheckboxMenuItem("Debug");
edit.add(debug);
test = new CheckboxMenuItem("Testing");
edit.add(test);
mbar.add(edit);
// create an object to handle action and item events
MyMenuHandler handler = new MyMenuHandler(this);
```

10.7.1.6. Dialog boxes

```
        // register it to receive those events
        item1.addActionListener(handler);
        item2.addActionListener(handler);
        item3.addActionListener(handler);
        item4.addActionListener(handler);
        item5.addActionListener(handler);
        item6.addActionListener(handler);
        item7.addActionListener(handler);
        item8.addActionListener(handler);
        item9.addActionListener(handler);
        item10.addActionListener(handler);
        debug.addItemListener(handler);
        test.addItemListener(handler);
        // create an object to handle window events
        MyWindowAdapter adapter = new MyWindowAdapter(this);
        // register it to receive those events
        addWindowListener(adapter);
    }
    public void paint(Graphics g) {
    g.drawString(msg, 10, 200);
    if(debug.getState())
    g.drawString("Debug is on.", 10, 220);
    else
    g.drawString("Debug is off.", 10, 220);
    if(test.getState())
    g.drawString("Testing is on.", 10, 240);
    else
    g.drawString("Testing is off.", 10, 240);
    }
}
```

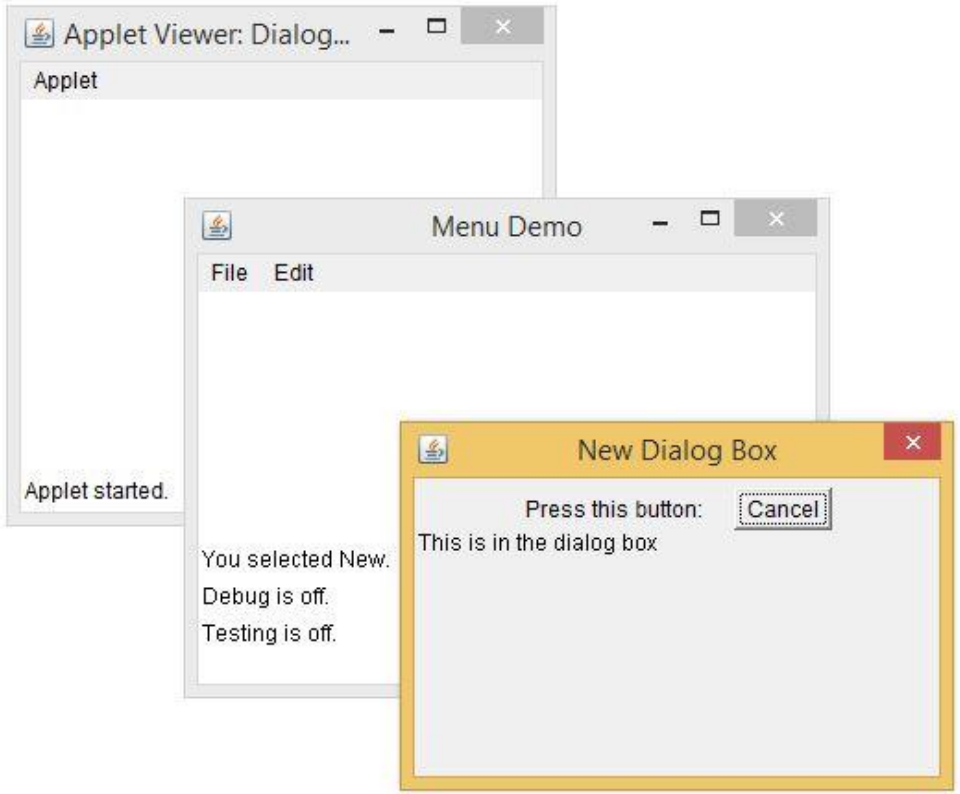
```
class MyWindowAdapter extends WindowAdapter {
    MenuFrame menuFrame;
    public MyWindowAdapter(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
    public void windowClosing(WindowEvent we) {
        menuFrame.dispose();
    }
}

class MyMenuHandler implements ActionListener, ItemListener {
    MenuFrame menuFrame;
    public MyMenuHandler(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
    // Handle action events.
    public void actionPerformed(ActionEvent ae) {
        String msg = "You selected ";
        String arg = ae.getActionCommand();
        // Activate a dialog box when New is selected.
        if(arg.equals("New...")) {
            msg += "New.";
            SampleDialog d = new
            SampleDialog(menuFrame, "New Dialog Box");
            d.setVisible(true);
        }
```

10.7.1.6. Dialog boxes

```
        // Try defining other dialog boxes for these options.
        else if(arg.equals("Open..."))
            msg += "Open.";
        else if(arg.equals("Close"))
            msg += "Close.";
        else if(arg.equals("Quit..."))
            msg += "Quit.";
        else if(arg.equals("Edit"))
            msg += "Edit.";
        else if(arg.equals("Cut"))
            msg += "Cut.";
        else if(arg.equals("Copy"))
            msg += "Copy.";
        else if(arg.equals("Paste"))
            msg += "Paste.";
        else if(arg.equals("First"))
            msg += "First.";
        else if(arg.equals("Second"))
            msg += "Second.";
        else if(arg.equals("Third"))
            msg += "Third.";
        else if(arg.equals("Debug"))
            msg += "Debug.";
        else if(arg.equals("Testing"))
            msg += "Testing.";
        menuFrame.msg = msg;
        menuFrame.repaint();
    }

    public void itemStateChanged(ItemEvent ie) {
        menuFrame.repaint();
    }
}
```

```
// Create frame window.
public class DialogDemo extends Applet {
    Frame f;
    public void init() {
        f = new MenuFrame("Menu Demo");
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));
        setSize(width, height);
        f.setSize(width, height);
        f.setVisible(true);
    }
    public void start() {
        f.setVisible(true);
    }
    public void stop() {
        f.setVisible(false);
    }
}
```

10.7.1.6. Dialog boxes



Here is sample output from the **DialogDemo** applet: