
UNIT 4 **GRAPHICS AND USER INTERFACES**

Structure	Page Nos.
4.0 Introduction	63
4.1 Objectives	63
4.2 Graphics Contexts and Graphics Objects	63
4.2.1 Color Control	
4.2.2 Fonts	
4.2.3 Coordinate System	
4.3 User Interface Components	70
4.4 Building User Interface with AWT	71
4.5 Swing-based GUI	76
4.6 Layouts and Layout Manager	77
4.7 Container	83
4.8 Summary	84
4.9 Solutions/Answer	85

4.0 INTRODUCTION

We all admire good graphics and animation on the internet. Java technology provides you the platform to develop these graphics using the Graphics class. In this Unit you have with an overview of several java capabilities for drawing two-dimensional shapes, colors and fonts. You will learn to make your own interface, which will be user-interactive and friendly. These interfaces can be made either using java AWT components or java SWING components. In this Unit you will learn to use some basic components like button, text field, text area, etc.

Interfaces using GUI allow users to spend less time trying to remember which keystroke sequences do what and allow them to spend more time using the program in a productive manner. It is very important to learn how you can beautify your components placed on the canvas area using FONT and COLOR class. For placing components on different layouts, knowing how to use various Layout managers is essential.

4.1 OBJECTIVES

After going through this Unit, you will be able to:

- explain the principles of graphical user interfaces;
- differentiate between AWT and SWING components;
- design a user friendly Interface;
- applying color to interfaces;
- using Font class in programs;
- use the Layout Manager, and
- use Container Classes.

4.2 GRAPHICS CONTEXTS AND GRAPHICS OBJECTS

How can you build your own animation using Graphics Class? A Java graphics context enables drawing on the screen. A Graphics object manages a graphics context by controlling how objects are drawn. Graphics objects contain methods for drawing, font manipulation, color manipulations, and the other related operations. It has been developed using the Graphics object g (the argument to the applet's paint method) to

manage the applet's graphics context. In other words, you can say that Java's Graphics is capable of:

- Drawing 2D shapes
- Controlling colors
- Controlling fonts
- Providing Java 2D API
- Using More sophisticated graphics capabilities
- Drawing custom 2D shapes
- Filling shapes with colors and patterns.

Before we begin drawing with Graphics, you must understand Java's coordinate system, which is a scheme for identifying every possible point on the screen. Let us start with Graphics Context and Graphics Class.

Graphics Context and Graphics Class

- Enables drawing on screen
- Graphics object manages graphics context
- Controls how objects are drawn
- Class Graphics is abstract
- Cannot be instantiated
- Contributes to Java's portability
- Class Component method paint takes Graphics object.

The *Graphics* class is the abstract base class for all graphics contexts. It allows an application to draw onto components that are realized on various devices, as well as on to off-screen images.

Public Abstract Class Graphics Extends Object

You have seen that every applet performs drawings on the screen.

Graphics Objects

In Java, all drawing takes place via a Graphics object. This is an instance of the class *java.awt.Graphics*.

Initially the Graphics object you use will be passed as an argument to an applet's paint() method. The drawing can be done by using Applet Panels, Frames, Buttons, Canvases, etc.

Each Graphics object has its own coordinate system, and methods for drawing strings, lines, rectangles, circles, polygons, etc. Drawing in Java starts with particular Graphics object. You get access to the Graphics object through the *paint(Graphics g)* method of your applet.

Each draw method call will look like

```
g.drawString("Hello World", 0, 50);
```

where g is the particular Graphics object with which you're drawing.

For convenience sake, in this Unit the variable g will always refer to a pre-existing object of the Graphics class. It is not a rule. You are free to use some other name for your particular Graphics context, such as myGraphics or applet-Graphics, or anything else.

4.2.1 Color Control

It is known that Color enhances the appearance of a program and helps in conveying meanings. To provide color to your objects use class Color, which defines methods and constants for the manipulation of colors. Colors are created from **red**, **green** and **blue** components **RGB** values.

All three RGB components can be integers in the range 0 to 255, or floating point values in the range 0.0 to 1.0

The first part defines the amount of *red*, the second defines the amount of *green* and the third defines the amount of *blue*. So, if you want to give a dark red color to your graphics you will have to give the first parameter value 255 and two parameter zero.

Some of the most common colors are available by name and their RGB values in *Table 1*.

Table 1: Colors and their RGB values

Color Constant	Color	RGB Values
Public final static Color ORANGE	Orange	255, 200, 0
Public final static Color PINK	Pink	255, 175, 175
Public final static Color CYAN	Cyan	0, 255, 255
Public final static Color MAGENTA	Magenta	255, 0, 255
Public final static Color YELLOW	Yellow	255, 255, 0
Public final static Color BLACK	Black	0, 0, 0
Public final static Color WHITE	White	255, 255, 255
Public final static Color GRAY	Gray	128, 128, 128
Public final static Color LIGHT_GRAY	light gray	192, 192, 192
Public final static Color DARK_GRAY	dark gray	64, 64, 64
Public final static Color RED	Red	255, 0, 0
Public final static Color GREEN	Green	0, 255, 0
Public final static Color BLUE	Blue	0, 0, 255

Color Methods

To apply color in your graphical picture (objects) or text, two Color methods, `getColor` and `setColor`, are provided. Method `getColor` returns a Color object representing the current drawing color, and method `setColor` used to sets the current drawing color.

Color constructors

`public Color(int r, int g, int b)`: Creates a color based on the values of red, green and blue components expressed as integers from 0 to 255.

`public Color (float r, float g, float b)` : Creates a color based on the values of on red, green and blue components expressed as floating-point values from 0.0 to 1.0.

Color methods:

`public int getRed()`: Returns a value between 0 and 255 representing the red content

`public int getGreen()`: Returns a value between 0 and 255 representing the green content

`public int getBlue()`: Returns a value between 0 and 255 representing the blue content.

Graphics Methods For Manipulating Colors

`public Color getColor ()`: Returns a Color object representing the current color for the graphics context.

`public void setColor (Color c)`: Sets the current color for drawing with the graphics context.

As you do with any variable, you should preferably give your colors descriptive names. For instance

```
Color medGray = new Color(127, 127, 127);
Color cream = new Color(255, 231, 187);
Color lightGreen = new Color(0, 55, 0);
```

You should note that `Color` is not a property of a particular rectangle, string or other object you may draw. `Color` is a part of the `Graphics` object that does the drawing. You change the color of your `Graphics` object and everything you draw from that point forward will be in the new color, at least until you change it again.

When an applet starts running, its color is set to *black by default*. You can change this to red by calling `g.setColor(Color.red)`. You can change it back to black by calling `g.setColor(Color.black)`.



Check Your Progress 1

- 1) What are the various color constructors?

.....

 ...

- 2) Write a program to set the color of a string to red.

.....

 ...

- 3) What is the method to retrieve the color of the text? Write a program to retrieve R G B values in a given color.

.....

Now we will discuss how different types of fonts can be provided to your strings.

4.2.2 Fonts

You must have noticed that until now all the applets have used the default font. However, unlike HTML Java allows you to choose your fonts. Java implementations are guaranteed to have a *serif* font like *Times* that can be accessed with the name "*Serif*", a monospaced font like *courier* that can be accessed with the name "*Mono*", and a *sans serif* font like *Helvetica* that can be accessed with the name "*SansSerif*".

How can you know the available fonts on your system for an applet program? You can list the fonts available on the system by using the `getFontList()` method from `java.awt.Toolkit`. This method returns an array of strings containing the names of the available fonts. These may or may not be the same as the fonts to installed on your system. Implementation is dependent on whether or not all the fonts in a system are available to the applet.

Choosing a font is very easy. You just create a new `Font` object and then call `setFont(Font f)`.

To instantiate a `Font` object, the constructor

```
public Font(String name, int style, int size)
```

can be used. *name* is the name of the font family, e.g. "Serif", "SansSerif", or "Mono".

size is the size of the font in points. In computer graphics, a point is considered to be equal to one pixel. 12 points is a normal size font.

style is mnemonic constant from `java.awt.Font` that tells whether the text will be bold, italics, or plain. The three constants are, `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC`.

In other words, The Class `Font` contains methods and constants for font control. `Font` constructor takes three arguments

Font name	Monospaced, SansSerif, Serif, etc.
Font style	<code>Font.PLAIN</code> , <code>Font.ITALIC</code> and <code>Font.BOLD</code>
Font size	Measured in points (1/72 of inch)

Graphics method for Manipulating Fonts

`public Font getFont()`: Returns a `Font` object reference representing the current `Font`.
`public void setFont(Font f)`: Sets the current font, style and size specified by the `Font` object reference `f`.

4.2.2.1 FontMetrics

Sometimes you will need to know how much space a particular string will occupy. You can find this out with a `FontMetrics` object.

`FontMetrics` allow you to determine the height, width or other useful characteristics of a particular string, character, or array of characters in a particular font. In *Figure 1*, you can see a string with some of its basic characteristics.

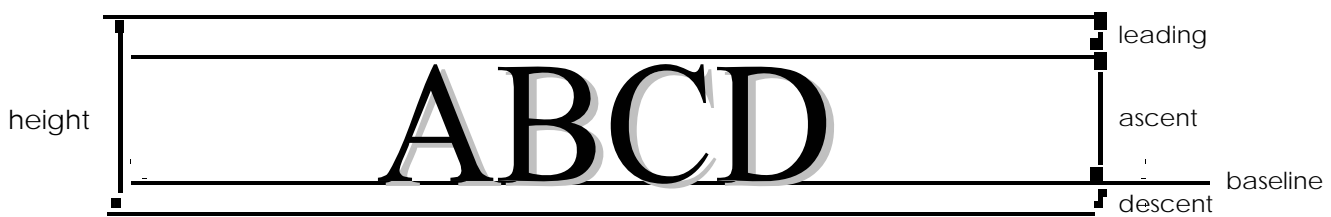


Figure 1: String Characteristics

In order to tell where and whether to wrap a `String`, you need to measure the string, not its length in characters, which can be variable in its width, and height in pixels. Measurements of this sort on strings clearly depend on the font that is used to draw the string. All other things being equal, a 14-point string will be wider than the same string in 12 or 10-point type.

To measure character and string sizes, you need to look at the `FontMetrics` of the current font. To get a `FontMetrics` object for the current `Graphics` object, you use the `java.awt.Graphics.getFontMetrics()` method.

`java.awt.FontMetrics` provide method `stringWidth(String s)` to return the width of a string in a particular font, and method `getLeading()` to get the appropriate line spacing for the font. There are many more methods in `java.awt.FontMetrics` that let you measure the height and width of specific

characters, as well as ascenders, descenders and more, but these three methods will be sufficient for basic programs.

Check Your Progress 2

- 1) Write a program to set the font of your String as font name as “Arial”,font size as 12 and font style as FONT.ITALIC.

.....

- 2) What is the method to retrieve the font of the text? Write a program for font retrieval.

.....

- 3) Write a program that will give you the Fontmetrics parameters of a String.

.....

Knowledge of the co-ordinate system is essential to play with positioning of objects in any drawing. Now, you will see how coordinates are used in java drawings.

4.2.3 Coordinate System

By Default, the upper left corner of a GUI component (such as applet or window) has the coordinates (0,0). A Coordinate pair is composed of x-coordinate (the horizontal coordinate) and a y-coordinate (the vertical coordinate). The x-coordinate is the horizontal distance moving right from the upper left corner.

The y-coordinate is the vertical distance moving down from the upper left corner. The x-axis describes every horizontal coordinate, and the y-axis describes every vertical coordinate. You must note that different display cards have different resolutions (i.e., the density of pixels varies). *Figure 2* represents coordinate system. This may cause graphics to appear to be different sizes on different monitors.

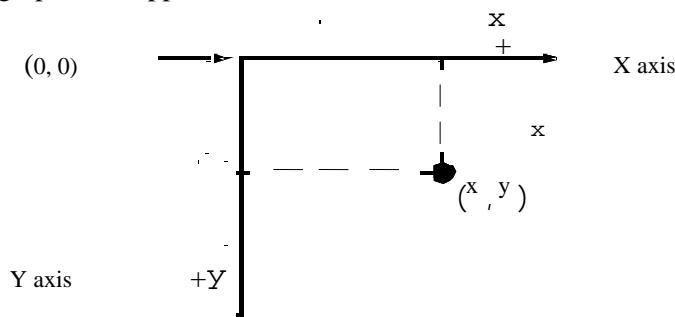


Figure 2: Co-ordinate System

Now we will move towards the drawing of different objects. In this section, we will demonstrate drawing in applications.

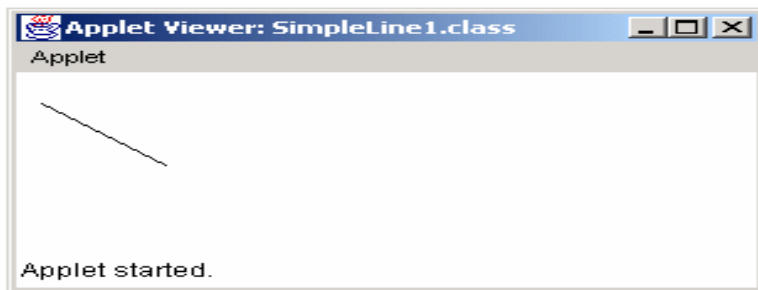
4.2.3.1 Drawing Lines

Drawing straight lines with Java can be done as follows:

call
`g.drawLine(x1,y1,x2,y2)`
 method, where $(x1, y1)$ and $(x2, y2)$ are the endpoints of your lines and `g` is the `Graphics` object you are drawing with. The following program will result in a line on the applet.

```
import java.applet.*;
import java.awt.*;
public class SimpleLine extends Applet
{
    public void paint(Graphics g)
    {
        g.drawLine(10, 20, 30, 40);
    }
}
```

Output:



4.2.3.2 Drawing Rectangles

Drawing rectangles is simple. Start with a `Graphics` object `g` and call its `drawRect()` method:

```
public void drawRect(int x, int y, int width, int height)
```

The first argument `int` is the left hand side of the rectangle, the second is the top of the rectangle, the third is the width and the fourth is the height. This is in contrast to some APIs where the four sides of the rectangle are given.

Remember that the upper left hand corner of the applet starts at $(0, 0)$, not at $(1, 1)$. This means that a 100 by 200 pixel applet includes the points with x coordinates between 0 and 99, not between 0 and 100. Similarly the y coordinates are between 0 and 199 inclusive, not 0 and 200.

4.2.3.3 Drawing Ovals and Circles

Java has methods to draw outlined and filled ovals. These methods are called `drawOval()`, and `fillOval()` respectively. These two methods are:

```
public void drawOval(int left, int top, int width, int height)
public void fillOval(int left, int top, int width, int height)
```

Instead of dimensions of the oval itself, the dimensions of the smallest rectangle, which can enclose the oval, are specified. The oval is drawn as large as it can be to touch the rectangle's edges at their centers. *Figure 3* may help you to understand properly.



Figure 3: An Oval

The arguments to `drawOval()` are the same as the arguments to `drawRect()`. The first int is the left hand side of the enclosing rectangle, the second is the top of the enclosing rectangle, the third is the width and the fourth is the height. There is no special method to draw a circle. Just draw an oval inside a square.

4.2.3.4 Drawing Polygons and Polylines

You have already seen that in Java rectangles are defined by the position of their upper left hand corner, their height, and their width. However, it is implicitly assumed that there is in fact an upper left hand corner. What's been assumed so far is that the sides of the rectangle are parallel to the coordinate axes. You cannot yet handle a rectangle that has been rotated at an arbitrary angle.

There are some other things you cannot handle either, e.g., triangles, stars, rhombuses, kites, octagons, and more. To take care of this broad class of shapes Java has a **Polygon** class.

Polygons are defined by their corners. No assumptions are made about them except that they lie in a 2-D plane. The basic constructor for the **Polygon** class is

```
public Polygon(int[] xpoints, int[] ypoints, int npoints)
```

xpoints is an array that contains the x coordinates of the polygon. **ypoints** is an array that contains the y coordinates. Both should have the length **npoints**. Thus to construct a right triangle with the right angle on the origin you would type

```
int[] xpoints = {0, 3, 0};
int[] ypoints = {0, 0, 4};
Polygon myTriangle = new Polygon(xpoints, ypoints, 3);
```

To draw the polygon, you can use `java.awt.Graphics`'s `drawPolygon(Polygon p)` method within your `paint()` method like this:

```
g.drawPolygon(myTriangle);
```

You can pass the arrays and number of points directly to the `drawPolygon()` method, if you prefer:

```
g.drawPolygon(xpoints, ypoints, xpoints.length);
```

There's also an overloaded `fillPolygon()` method. You can call this method like

```
g.fillPolygon(myTriangle);
g.fillPolygon(xpoints, ypoints, xpoints.length());
```

To simplify user interaction and make data entry easier, Java provides different controls and interfaces. Now, let us see some of the basic user interface components of Java.

4.3 USER INTERFACE COMPONENTS

Java provides many controls. Controls are components, such as buttons, labels and text boxes that can be added to containers like frames, panels and applets. The `Java.awt` package provides an integrated set of classes to manage user interface components.

Components are placed on the user interface by adding them to a container. A container itself is a component. The easiest way to demonstrate interface design is by using the container you have been working with, i.e., the `Applet` class. The simplest form of Java AWT component is the basic *User Interface Component*. You can create and add these to your applet without any need to know anything about creating containers or panels. In fact, your applet, even before you start painting and drawing and handling events, is an AWT container. Because an applet is a container, you can put any of AWT components, and (or) other containers, in it.

In the next section of this Unit, you will learn about the basic User Interface components (controls) like labels, buttons, check boxes, choice menus, and text fields. You can see in Table 2a, a list of all the Controls in Java AWT and their respective functions. In Table 2b, a list of classes for these control are given.

4.4 BUILDING USER INTERFACE WITH AWT

In order to add a control to a container, you need to perform the following two steps:

1. Create an object of the control by passing the required arguments to the constructor.
2. Add the component (control) to the container.

Table 2a: Controls in Java

CONTROLS	FUNCTIONS
Textbox	Accepts single line alphanumeric entry.
TextArea	Accepts multiple line alphanumeric entry.
Push button	Triggers a sequence of actions.
Label	Displays Text.
Check box	Accepts data that has a yes/no value. More than one checkbox can be selected.
Radio button	Similar to check box except that it allows the user to select a single option from a group.
Combo box	Displays a drop-down list for single item selection. It allows new value to be entered.
List box	Similar to combo box except that it allows a user to select single or multiple items. New values cannot be entered.

Table 2b: Classes for Controls

CONTROLS	CLASS
Textbox	TextField
TextArea	TextArea
Push button	Button
Check box	CheckBox
Radio button	CheckboxGroup with CheckBox
Combo box	Choice
List box	List

The Button

Let us first start with one of the simplest of UI components: the button. Buttons are used to trigger events in a GUI environment (we have discussed Event Handling in detail in the previous Unit: Unit 1 Block 4 of this course). The Button class is used to create buttons. When you add components to the container, you do not specify a set of coordinates that indicate where the components are to be placed. A layout manager

for the container handles the arrangement of components. The default layout for a container is a flow layout (for an applet, also, default layout will be the flow layout). You will learn more about different layouts in a later section of this Unit. Now let us write a simple code to test our button class.

To create a button, use one of the following constructors:

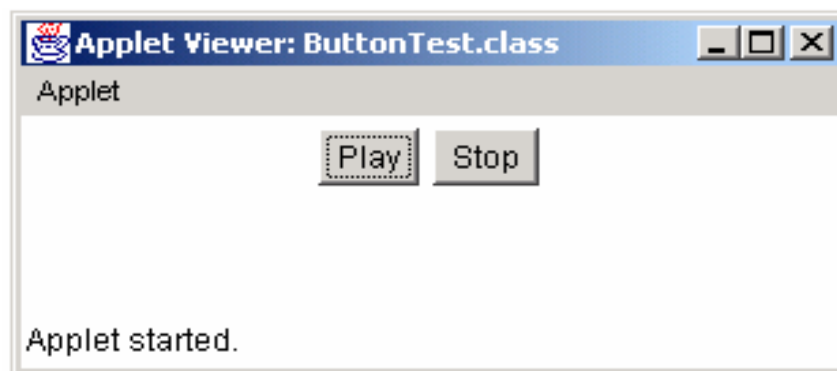
Button() creates a button with no text label.

Button(String) creates a button with the given string as label.

Example:

```
/*
<Applet code= "ButtonTest.class"
Width = 500
Height = 100>
</applet>
*/
import java.awt.*;
import java.applet.Applet;
public class ButtonTest extends Applet
{
    Button b1 = new Button ("Play");
    Button b2 = new Button ("Stop");
    public void init(){
        add(b1);
        add(b2);
    }
}
```

Output:



As you can see this program will place two buttons on the Applet with the captions Play and Stop.

The Label

Labels are created using the Label class. Labels are basically used to identify the purpose of other components on a given interface; they cannot be edited directly by the user. Using a label is much easier than using a drawString() method because labels are drawn automatically and don't have to be handled explicitly in the paint() method. Labels can be laid out according to the layout manager, instead of using [x, y] coordinates, as in drawString().

To create a Label, use any one of the following constructors:

Label(): creates a label with its string aligned to the left.

Label(String): creates a label initialized with the given string, and aligned left.

Label(String, int): creates a label with specified text and alignment indicated by any one of the three int arguments. Label.Right, Label.Left and Label.Center.

getText() method is used to indicate the current label's text setText() method to change the label's and text. setFont() method is used to change the label's font.

The Checkbox

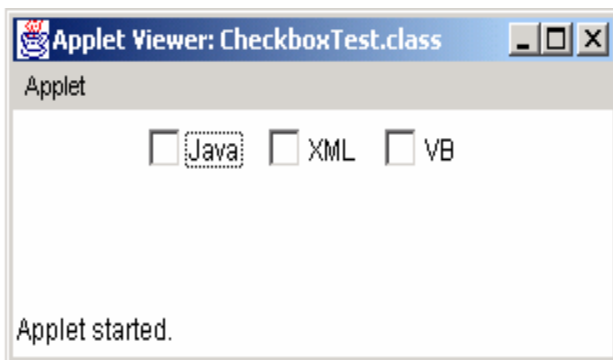
Check Boxes are labeled or unlabeled boxes that can be either "Checked off" or "Empty". Typically, they are used to select or deselect an option in a program.

Sometimes Check Boxes are *nonexclusive*, which means that if you have six check boxes in a container, all the six can either be checked or unchecked at the same time. This component can be organized into a Check Box Group, which is sometimes called radio buttons. Both kinds of check boxes are created using the Checkbox class. To create a nonexclusive check box you can use one of the following constructors: Checkbox() creates an unlabeled checkbox that is not checked. Checkbox(String) creates an unchecked checkbox with the given label as its string.

After you create a checkbox object, you can use the setState(boolean) method with a true value as argument for checked checkboxes, and false to get unchecked. Three checkboxes are created in the example given below, which is an applet to enable you to select up to three courses at a time.

```
import java.awt.*;
public class CheckboxTest extends java.applet.Applet
{
    Checkbox c1 = new Checkbox ("Java");
    Checkbox c2 = new Checkbox ("XML");
    Checkbox c3 = new Checkbox ("VB");
    public void init(){
        add(c1);
        add(c2);
        add(c3);
    }
}
```

Output:



The Checkbox group

CheckboxGroup is also called a radio button, or exclusive check boxes. To organize several Checkboxes into a group so that only one can be selected at a time, you can create CheckboxGroup object as follows:

```
CheckboxGroup radio = new CheckboxGroup ();
```

The CheckboxGroup keeps track of all the check boxes in its group. We have to use this object as an extra argument to the *Checkbox* constructor.

Checkbox (String, CheckboxGroup, Boolean) creates a checkbox labeled with the given string that belongs to the CheckboxGroup indicated in the second argument. The last argument equals true if the box is checked and false otherwise.

The set Current (checkbox) method can be used to make the set of currently selected

check boxes in the group. There is also a `get Current ()` method, which returns the currently selected checkbox.

The Choice List

Choice List is created from the Choice class. List has components that enable a single item to be picked from a pull-down list. We encounter this control very often on the web when filling out forms.

The first step in creating a Choice

You can create a choice object to hold the list, as shown below:

```
Choice cgender = new Choice();
```

Items are added to the Choice List by using `addItem(String)` method the object. The following code adds two items to the gender choice list.

```
cgender.addItem("Female");
cgender.addItem("Male");
```

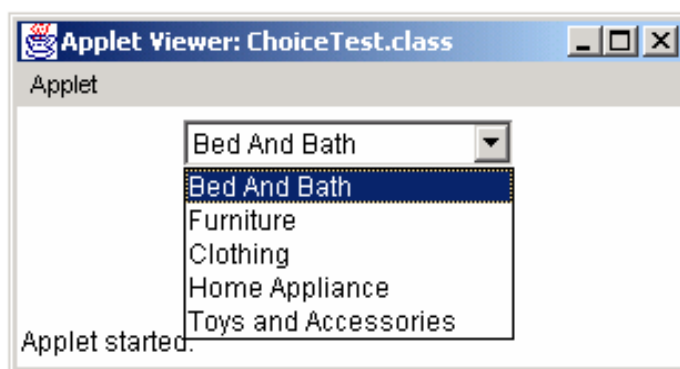
After you add the Choice List, it is added to the container like any other component using the `add()` method.

The following example shows an Applet that contains a list of shopping items in the store.

```
import java.awt.*;
Public class ChoiceTest extends java.applet.Applet
{
Choice shoplis = new Choice();

Public void init(){
shoplis.addItem("Bed And Bath");
shoplis.addItem("Furniture");
shoplis.addItem("Clothing");
shoplis.addItem("Home Appliance");
shoplis.addItem("Toys and Accessories");
add(shoplis);
}
```

Output:



The choice list class has several methods, which are given in *Table 3*.

Table 3: Choice List Class Methods

Method	Action
<code>getItem()</code>	Returns the string item at the given position (items inside a choice begin at 0, just like arrays)
<code>countItems()</code>	Returns the number of items in the menu

<code>getSelectedIndex()</code>	Returns the index position of the item that's selected
<code>getSelectedItem()</code>	Returns the currently selected item as a string
<code>select(int)</code>	Selects the item at the given position
<code>select(String)</code>	Selects the item with the given string

The Text Field

To accept textual data from user, AWT provided two classes, *TextField* and *TextArea*. *TextField* handles a single line of text and does not have scrollbars, whereas the *TextArea* class handles multiple lines of text. Both the classes are derived from the *TextComponent* class, hence, they share many common methods. *TextFields* provide an area where you can enter and edit a single line of text. To create a text field, use one of the following constructors:

`TextField()`: creates an empty *TextField* with no specified width.

`TextField(int)`: creates an empty text field with enough width to display the specified number of characters (this has been depreciated in Java2).

`TextField(String)`: creates a text field initialized with the given string.

`TextField(String, int)`: creates a text field with specified text, and specified width.

For example, the following line creates a text field 25 characters wide with the string "Brewing Java" as its initial contents:

```
TextField txtfld = new TextField ("Brewing Java", 25);
add(txtfld);
```

TextField, can use methods like:

`setText()`: Used to set the text in text field.

`getText()`: Used to get the text currently contained by text field.

`setEditable()`: Used to provide control whether the content of text field may be modified by user or not.

`isEditable()`: It returns **true** if the text in text filed may be changed and **false** otherwise.

Text Area

The *TextArea* is an editable text field that can handle more than one line of input.

Text areas have horizontal and vertical scrollbars to scroll through the text. Adding a text area to a container is similar to adding a text field. To create a text area, you can use one of the following constructors:

`TextArea()`: creates an empty text area with unspecified width and height.

`TextArea(int, int)`: creates an empty text area with the indicated number of lines and specified width in characters.

`TextArea(String)`: creates a text area, initialized with the given string.

`TextField(String, int, int)`: creates a text area containing the indicated text and specified number of lines, and width in the characters.

The *TextArea*, similar to *TextField*, can use methods like `setText()`, `getText()`, `setEditable()`, and `isEditable()`.

In addition, there are two more methods like these. The first is the `insertText(String, int)` method, used to insert indicated strings at the character index specified by the second argument. The next one is `replaceText(String, int, int)` method, used to replace text between given integer position specified by second and third argument with the indicated string.

The basic idea behind the AWT is that a graphical Java program is a set of nested components, starting from the outermost window all the way down to the smallest UI component. Components can include things you can actually see on the screen, such as windows, menu bars, buttons, and text fields, and they can also include containers, which in turn, can contain other components.

I hope you have got a clear picture of Java AWT and its some basic UI components, In the next section of the Unit, we will deal with more advanced user interface components.

4.5 SWING - BASED GUI

You must be thinking that when you can make GUI interface with AWT package then what is the purpose of learning Swing-based GUI? Actually, Swing has *lightweight* components and does not write itself to the screen, but redirects it to the component it builds on. On the other hand, AWT is a heavyweight and has its own view port which sends the output to the screen. Heavyweight components also have their own

z-ordering (look and feel) which are a dependent on the machine on which the program is running. This is the reason why you cannot combine AWT and Swing in the same container. If you do, AWT will always be drawn on top of the Swing components.

Another difference is that Swing is pure Java, and therefore platform independent. Swing looks identical on all platforms, while AWT looks different on different platforms.

Basically, Swing provides a rich set of GUI components. Its features include model-UI separation, and a pluggable look and feel. Actually, you can make your GUI also with AWT, but with Swing, you can make it more user-friendly and interactive. Swing components make programs efficient.

Swing GUI components are packaged into Package javax.swing. In the Java class hierarchy there is a class

Class Component which contains a method paint for drawing Component onscreen

Class Container which is a collection of related components and contains method add for adding components and Class JComponent which has

Pluggable look and feel for customizing look and feel

Shortcut keys (*mnemonics*)

Common event-handling capabilities

The Hierarchy is as follows:

Object----→ Component--→ Container---→ JComponent

In Swings, we have classes prefixed with the letter 'J' like

JLabel -> Displays single line of read only text

JTextField -> Displays or accepts input in a single line

JTextArea -> Displays or accepts input in multiple lines

JCheckBox -> Gives choices for multiple options

JButton -> Accepts command and does the action

JList -> Gives multiple choices and display for selection

JRadioButton -> Gives choices for multiple option, but can select one at a time.



Check Your Progress 3

- 1) Write a program which draws a line, a rectangle, and an oval on the applet.

.....

- 2) Write a program that draws a color-filled line, a color-filled rectangle, and a color filled oval on the applet.

.....

- 3) Write a program to add various checkboxes under the CheckboxGroup

.....

- 4) Write a program in which the Applet displays a text area that is filled with a string, when the programs begin running.

.....

- 5) Describe the features of the Swing components that subclass J Component. What are the difference between Swing and AWT?

.....

Now, we will discuss different layouts to represent components in a container.

4.6 LAYOUTS AND LAYOUT MANAGER

When you add a component to an applet or a container, the container uses its *layout manager* to decide where to put the component. Different `LayoutManager` classes use different rules to place components.

`java.awt.LayoutManager` is an interface. Six classes in the java packages implement it:

- 1) `FlowLayout`
- 2) `BorderLayout`
- 3) `CardLayout`
- 4) `GridLayout`
- 5) `GridBagLayout`
- 6) plus `javax.swing.BoxLayout`

FlowLayout

A `FlowLayout` arranges widgets from left to right until there's no more space left. Then it begins a row lower, and moves from left to right again. Each component in a `FlowLayout` gets as much space as it needs, and no more.

This is the *default LayoutManager* for applets and panels. `FlowLayout` is the default layout for `java.awt.Panel` of which `java.applet.Applet` is a subclass. Therefore, you do not need to do anything special to create a `FlowLayout` in an applet. However you do need to use the following constructors if you want to use a

`FlowLayout` in a `Window`. `LayoutManagers` have constructors, like any other class.

The constructor for a `FlowLayout` is

```
public FlowLayout()
```

Thus, to create a new `FlowLayout` object you write

```
FlowLayout fl;
```

```
fl = new FlowLayout();
```

As usual, this can be shortened to

```
FlowLayout fl = new FlowLayout();
```

You tell an applet to use a particular `LayoutManager` instance by passing the object to the applet's `setLayout()` method like this: `this.setLayout(fl);`

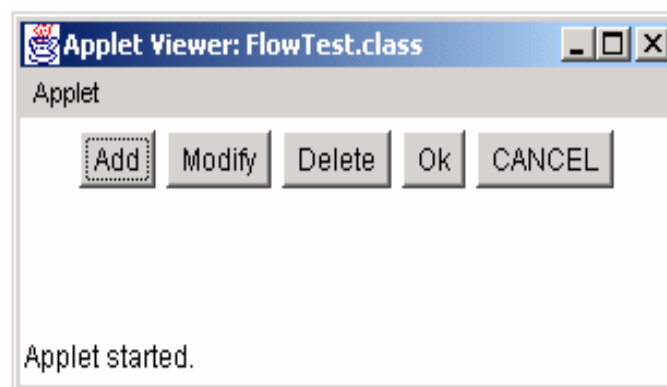
Most of the time `setLayout()` is called in the `init()` method. Normally, you just create the `LayoutManager` right inside the call to `setLayout()` like this:

```
this.setLayout(new FlowLayout());
```

For example, the following applet uses a `FlowLayout` to position a series of buttons that mimic the buttons on a tape deck:

```
import java.applet.*;
import java.awt.*;
public class FlowTest extends Applet {
    public void init() {
        this.setLayout(new FlowLayout());
        this.add( new Button("Add"));
        this.add( new Button("Modify"));
        this.add( new Button("Delete"));
        this.add( new Button("Ok"));
        this.add( new Button("CANCEL"));
    }
}
```

Output:



You can change the alignment of a `FlowLayout` in the constructor. Components are normally centered in an applet. You can make them left or right justified. To do this, just pass one of the defined constants `FlowLayout.LEFT`, `FlowLayout.RIGHT` or `FlowLayout.CENTER` to the constructor, e.g., `this.setLayout(new FlowLayout(FlowLayout.LEFT));`

Another constructor allows you to use the spacing option in `FlowLayout`:

```
public FlowLayout(int alignment, int horizontalSpace, int verticalSpace);
```

For instance to set up a `FlowLayout` with a 10 pixel horizontal gap and a 20 pixel vertical gap, aligned with the left edge of the panel, you would use the constructor `FlowLayout fl = new FlowLayout(FlowLayout.LEFT, 20, 10);`

The buttons are arranged according to a center-aligned `FlowLayout` with a 20 pixel horizontal spacing and a 10 pixel vertical spacing

BorderLayout

A `BorderLayout` organizes an applet into North, South, East, West, and Center sections. North, South, East, and West are the rectangular edges of the applet. They're continually resized to fit the sizes of the widgets included in them. Center is whatever is left over in the middle.

A `BorderLayout` places objects in the North, South, East, West, and Center of an applet. You create a new `BorderLayout` object much like a `FlowLayout` object, in the `init()` method call to `setLayout` like this:

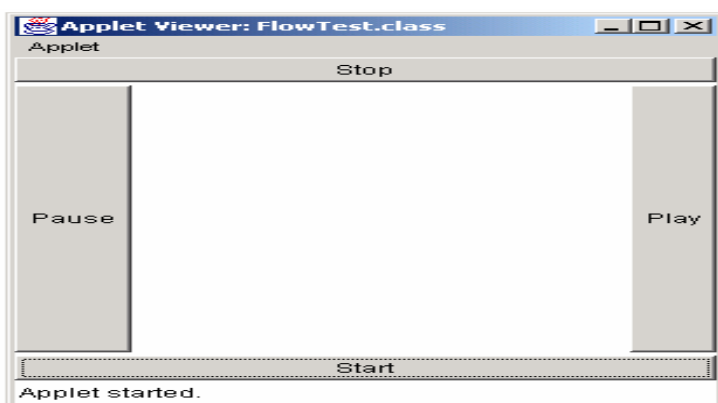
```
this.setLayout(new BorderLayout());
```

There's no centering, left alignment, or right alignment in a `BorderLayout`. However, you can add horizontal and vertical gaps between the areas. Here is how you would add a two pixel horizontal gap and a three pixel vertical gap to a `BorderLayout`:

```
this.setLayout(new BorderLayout(2,3));
```

To add components to a `BorderLayout`, include the name of the section you wish to add them to, as given below.

```
this.add("South", new Button("Start"));
import java.applet.*;
import java.awt.*;
public class BorderLayouttest extends Applet
{
public void init() {
this.setLayout(new BorderLayout(2, 3));
this.add("South", new Button("Start"));
this.add("North", new Button("Stop"));
this.add("East", new Button("Play"));
this.add("West", new Button("Pause"));
}
}
```



Card Layout

A `CardLayout` breaks the applet into a deck of cards, each of which has its own Layout Manager. Only one card appears on the screen at a time. The user flips between cards, each of which shows a different set of components. The common analogy is with HyperCard on the Mac and Tool book on Windows. In Java this might be used for a series of data input screens, where more input is needed than will comfortably fit on a single screen.

Grid Layout

A `GridLayout` divides an applet into a specified number of rows and columns, which form a grid of cells, *each equally sized and spaced*. It is important to note that each is equally sized and spaced as there is similarly named Layout known as `GridBagLayout`. As Components are added to the layout, they are placed in the cells,

starting at the upper left hand corner and moving to the right and down the page. Each component is sized to fit into its cell. This tends to squeeze and stretch components unnecessarily.

You will find the `GridLayout` is great for arranging Panels. A `GridLayout` specifies the number of rows and columns into which components will be placed. The Applet is broken up into a table of equal sized cells.

`GridLayout` is useful when you want to place a number of similarly sized objects. It is great for putting together lists of checkboxes and radio buttons as you did in the `Ingredients` Applet. `GridLayout` looks like *Figure 3*.



Figure 3: `GridLayout` Demo

Grid Bag Layout

`GridBagLayout` is the most precise of the five AWT Layout Managers. It is similar to the `GridLayout`, but components do not need to be of the same size. Each component can occupy one or more cells of the layout. Furthermore, components are not necessarily placed in the cells beginning at the upper left-hand corner and moving to the right and below.

In simple Applets with just a few components, you often need only one layout manager. In more complicated applets, however, you will often split your applet into panels, lay out the panels according to a layout manager, and give each panel its own layout manager that arranges the components inside it.

The `GridBagLayout` constructor is trivial, `GridBagLayout()` with no arguments.

```
GridBagLayout gbl = new GridBagLayout();
```

Unlike the `GridLayout()` constructor, this does not say how many rows or columns there will be. The cells your program refers to determine this. If you put a component in row 8 and column 2, then Java will make sure there are at least 9 rows and 3 columns. Rows and columns start counting at zero. Later, if you put a component in row 10 and column 4, Java will add the necessary extra rows and columns. You may have a picture in your mind of the finished grid, but Java does not need to know this when you create a `GridBagLayout`.

Unlike most other `LayoutManagers`, you should not create a `GridBagLayout` inside a cell to `setLayout()`. You will need access to the `GridBagLayout` object later in the applet when you use `GridBagConstraints`.

A **`GridBagConstraints`** object specifies the location and area of the component's display area within the container (normally the applet panel), and how the component is laid out inside its display area. The `GridBagConstraints`, in conjunction with the component's minimum size and the preferred size of the component's container, determines where the display area is placed within the applet.

The `GridBagConstraints()` constructor is trivial

```
GridBagConstraints gbc = new GridBagConstraints();
```

Your interaction with a `GridBagConstraints` object takes place through its 11 fields and 15 mnemonic constants.

gridx and gridy

The `gridx` and `gridy` fields specify the x and y coordinates of the cell at the upper left of the Component's display area. The upper-left-most cell has coordinates (0, 0). The mnemonic constant `GridBagConstraints.RELATIVE` specifies that the Component is placed immediately to the right of (`gridx`), or immediately below (`gridy`) the previous Component added to this container.

Gridwidth and Gridheight

The `gridwidth` and `gridheight` fields specify the number of cells in a row (`gridwidth`) or column (`gridheight`) in the Component's display area. The mnemonic constant `GridBagConstraints.REMAINDER` specifies that the Component should use all remaining cells in its row (for `gridwidth`) or column (for `gridheight`). The mnemonic constant `GridBagConstraints.RELATIVE` specifies that the Component should fill all but the last cell in its row (`gridwidth`) or column (`gridheight`).

Fill

The `GridBagConstraints.fill` field determines whether and how a component is resized if the component's display area is larger than the component itself. The mnemonic constants you use to set this variable are
`GridBagConstraints.NONE`: Don't resize the component
`GridBagConstraints.HORIZONTAL`: Make the component wide enough to fill the display area, but don't change its height.
`GridBagConstraints.VERTICAL`: Make the component tall enough to fill its display area, but don't change its width.
`GridBagConstraints.BOTH`: Resize the component enough to completely fill its display area both vertically and horizontally.

Ipadx and Ipady

Each component has a minimum width and a minimum height, smaller than which it will not be. If the component's minimum size is smaller than the component's display area, then only part of the component will be shown.

The `ipadx` and `ipady` fields let you increase this minimum size by padding the edges of the component with extra pixels. For instance setting `ipadx` to 2 will guarantee that the component is at least 4 pixels wider than its normal minimum. (`ipadx` adds 2 pixels to each side.)

Insets

The `insets` field is an instance of the `java.awt.Insets` class. It specifies the padding between the component and the edges of its display area.

Anchor

When a component is smaller than its display area, the `anchor` field specifies where to place it in the grid cell. The mnemonic constants you use for this purpose are similar to those used in a `BorderLayout`, but a little more specific.

They are

```
GridBagConstraints.CENTER
```

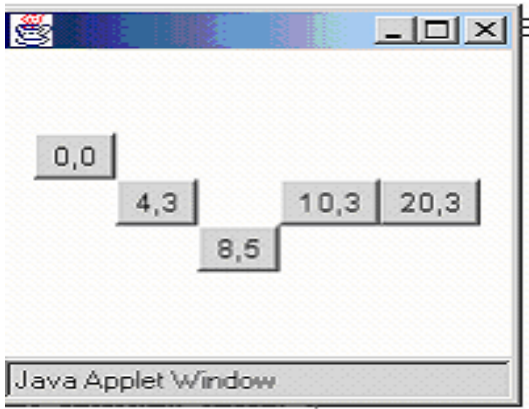
```
GridBagConstraints.NORTH
GridBagConstraints.NORTHEAST
GridBagConstraints.EAST
GridBagConstraints.SOUTHEAST
GridBagConstraints.SOUTH
GridBagConstraints.SOUTHWEST
GridBagConstraints.WEST
GridBagConstraints.NORTHWEST
The default is GridBagConstraints.CENTER.
```

weightx and weighty

The `weightx` and `weighty` fields determine how the cells are distributed in the container when the total size of the cells is less than the size of the container. With weights of zero (the default), the cells all have the minimum size they need, and everything clumps together in the center. All the extra space is pushed to the edges of the container. It doesn't matter where they go, and the default of center is fine. See the program given below for visualizing GridbagLayout.

```
import java.awt.*;
public class GridbagLayouttest extends Frame
{
    Button b1,b2,b3,b4,b5;
    GridBagLayout gbl=new GridBagLayout();
    GridBagConstraints gbc=new GridBagConstraints();
    public GridbagLayouttest()
    {
        setLayout(gbl);
        gbc.gridx=0;
        gbc.gridy=0;
        gbl.setConstraints(b1=new Button("0,0"),gbc);
        gbc.gridx=4;    //4th column
        gbc.gridy=3;    //3rd row
        gbl.setConstraints(b2=new Button("4,3"),gbc);
        gbc.gridx=8;    //8th column
        gbc.gridy=5;    //5rd row
        gbl.setConstraints(b3=new Button("8,5"),gbc);
        gbc.gridx=10;   //10th column
        gbc.gridy=3;    //3rd row
        gbl.setConstraints(b4=new Button("10,3"),gbc);
        gbc.gridx=20;   //20th column
        gbc.gridy=3;    //3rd row
        gbl.setConstraints(b5=new Button("20,3"),gbc);
        add(b1);
        add(b2);
        add(b3);
        add(b4);
        add(b5);
        setSize(200,200);
        setVisible(true);
    }
    public static void main(String a[])
    {
        GridbagLayouttest gb= new GridbagLayouttest();
    }
}
```

Output:



Now, we will discuss different containers.

4.7 CONTAINER

You might think that a container will be a thing that contains something, like a bowl. Then you are right!! Actually, a container object is derived from the `java.awt.Container` class and is one of (or inherited from) three primary classes: `java.awt.Window`, `java.awt.Panel`, `java.awt.ScrollPane`.

The `Window` class represents a stand alone window, either an application window in the form of a `java.awt.Frame`, or a dialog box in the form of a `java.awt.Dialog`. The `java.awt.Panel` class is not a standalone window by itself; instead, it acts as a background container for all other components on a form. For instance, the `java.awt.Applet` class is a direct descendant of `java.awt.Panel`.

The three steps common for all Java GUI applications are:

1. Creation of a container
2. Layout of GUI components
3. Handling of events.

The `Container` class contains the `setLayout()` method so that you can set the default `LayoutManager` to be used by your GUI. To actually add components to the container, you can use the container's `add()` method:

```
Panel p = new java.awt.Panel();
Button b = new java.awt.Button("OK");
p.add(b);
```

A `JPanel` is a `Container`, which means that it can contain other components. GUI design in Java relies on a layered approach where each layer uses an appropriate layout manager.

`FlowLayout` is the default for `JPanel` objects. To use a different manager, use either of the following:

```
JPanel pane2 = new JPanel()           // make the panel first
pane2.setLayout(new BorderLayout()); // then reset its manager
JPanel pane3 = new JPanel(new BorderLayout()); // all in one!
```

Check Your Progress 4

- 1) Why do you think Layout Manager is important?

.....

.....

.....

.....

...

- 2) How does the repaint() method work with Applet?

.....
.....
.....
.....
...

- 3) Each type of container comes with a default layout manager. Default for a Frame, Window or Dialog is _____

- 4) Give examples of stretchable components and non-stretchable components.

.....
.....
.....
.....
...

- 5) The Listener interfaces inherit directly from which package.

.....
.....
.....
.....
...

- 6) How many Listeners are there for trapping mouse movements?

.....
.....
.....
.....
...

4.8 SUMMARY

This Unit is designed for you to learn about the graphical interfaces in Java. In this Unit you become familiar with the AWT and SWING packages, which are used to add components in the container or a kind of tray. Swings are lightweight components and more flexible as compared to AWT. You learned to beautify your text by using Font class and Color class. For Geometric figures various in built classes of Java are used for drawing lines, circles, polygons etc. are used. You learned to place the components in various layouts. Java provides various layouts, some of which are: FlowLayout, GridBagLayout, GridLayout, CardLayout, BorderLayout.

4.9 SOLUTIONS/ANSWERS

Check Your Progress 1

- 1) Java defines two constructors for the Color Class, of which, one constructor takes three integer arguments, and another takes three float arguments.

```
public Color (int r, int g, int b)
```

Creates a color based on red, green and blue components expressed as integers from 0 to 255. Here, the r, g, b means red, green and blue contents respectively.

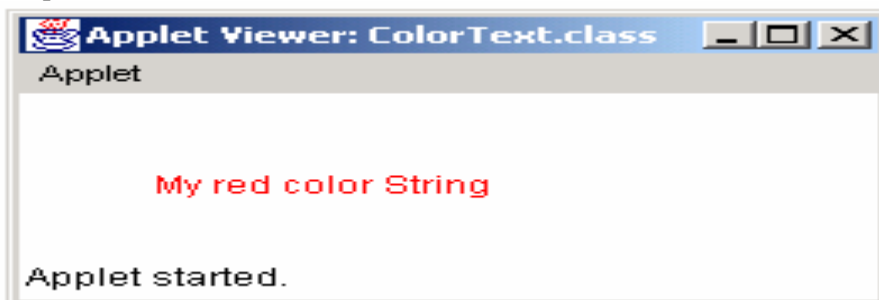
```
public Color(float r, float g, float b)
```

Creates a color based on red, green and blue components expressed as floating-point values from 0.0 to 1.0. Here, the r, g, b means red, green and blue contents respectively.

2)

```
import java.awt.*;
import java.applet.Applet;
public class ColorText extends Applet
{
    public void paint(Graphics g)
    {
        g.setColor(Color.red);
        g.drawString("My Red color String",40,50);
    }
}
```

Output:



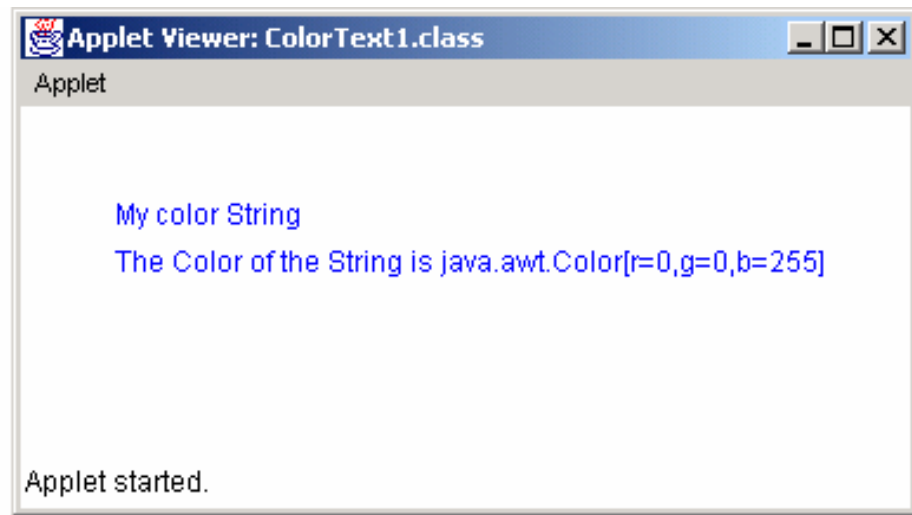
- 3) If you want to retrieve the color of a text or a String you can use the method `public int getColor()`. This returns a value between 0 and 255 representing the color content.

The Program given below is written for getting RGB components of the color

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

```
import java.applet.Applet;
public class ColorText extends Applet
{
    public void paint(Graphics g)
    {
        int color;
        g.drawString("My Pink color String",40,50);
        color = getColor(g);
        g.drawString("The Color of the String is "+color ,
        40,35);
    }
}
```

Output:



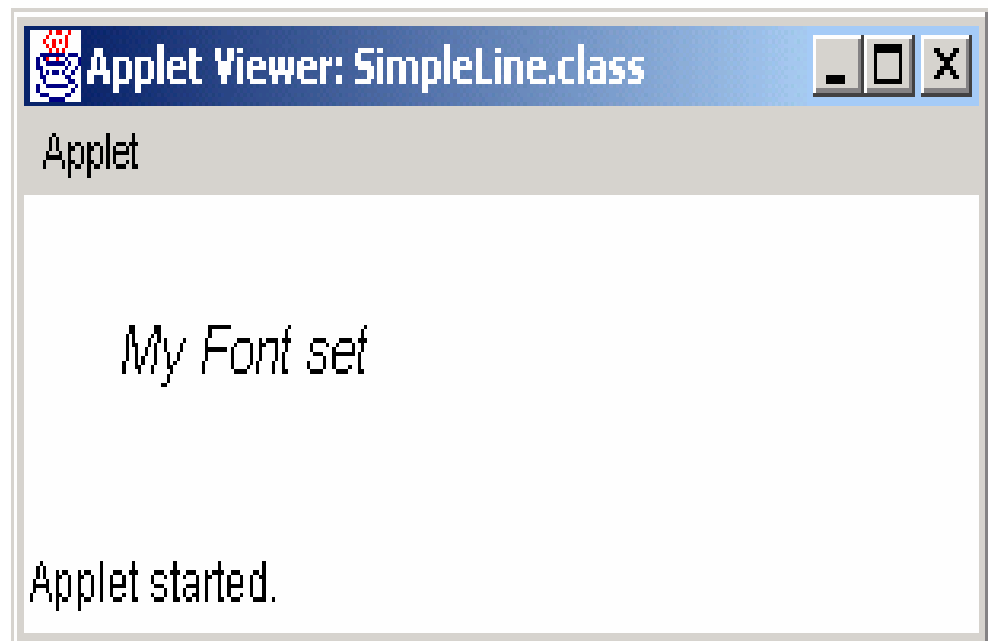
Check Your Progress 2

```

1)
import java.applet.*;
import java.awt.*;
public class SimpleLine extends Applet
{
    public void paint(Graphics g)
    {
        g.drawString("My Font set" , 30, 40);
        g.setFont("Arial",Font.ITALIC,15);
    }
}

```

Output:



2) To retrieve the font of the string, the method **getFont()** is used.

Program for font retrieval:

```

import java.applet.*;
import java.awt.*;
public class SimpleLine extends Applet
{
    public void paint(Graphics g)
    {

```

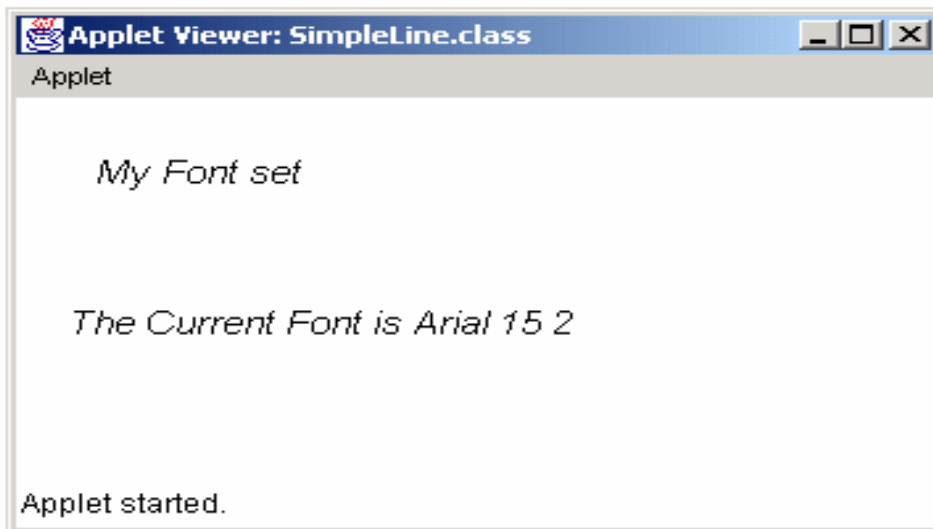


```

g.drawString("My Font set" , 30, 40);
g.setFont("Arial",Font.ITALIC,15);
g.drawString("current Font is " + g.getFont(),40,50);
g.drawString( g.getFont().getName() + " " +
g.getFont().getSize()+" " +g.getFont().getStyle(), 20,
110 );
}
}

```

Output:



Note: The `getStyle()` method returns the integer value e.g. above value 2 represent ITALIC.

3)

```

import java.awt.*;
import java.applet.Applet;
public class Metrics extends Applet {
    public void paint(Graphics g)
    {
        g.drawString("Hello How are You",50,60);
        g.setFont( new Font( "SansSerif", Font.BOLD, 12 ) );
        FontMetrics metrics = g.getFontMetrics();
        g.drawString( "Current font: " + g.getFont(), 10, 40 );
        g.drawString( "Ascent: " + metrics.getAscent(), 10, 55 );
        g.drawString( "Descent: " + metrics.getDescent(), 10, 70 );
        g.drawString( "Height: " + metrics.getHeight(), 10, 85 );
        g.drawString( "Leading: " + metrics.getLeading(), 10, 100 );
        Font font = new Font( "Serif", Font.ITALIC, 14 );
        metrics = g.getFontMetrics( font );
        g.setFont( font );
        g.drawString( "Current font: " + font, 10, 130 );
        g.drawString( "Ascent: " + metrics.getAscent(), 10, 145 );
        g.drawString( "Descent: " + metrics.getDescent(), 10, 160 );
        g.drawString( "Height: " + metrics.getHeight(), 10, 175 );
        g.drawString( "Leading: " + metrics.getLeading(), 10, 190 );
    } // end method paint
}

```

Output:

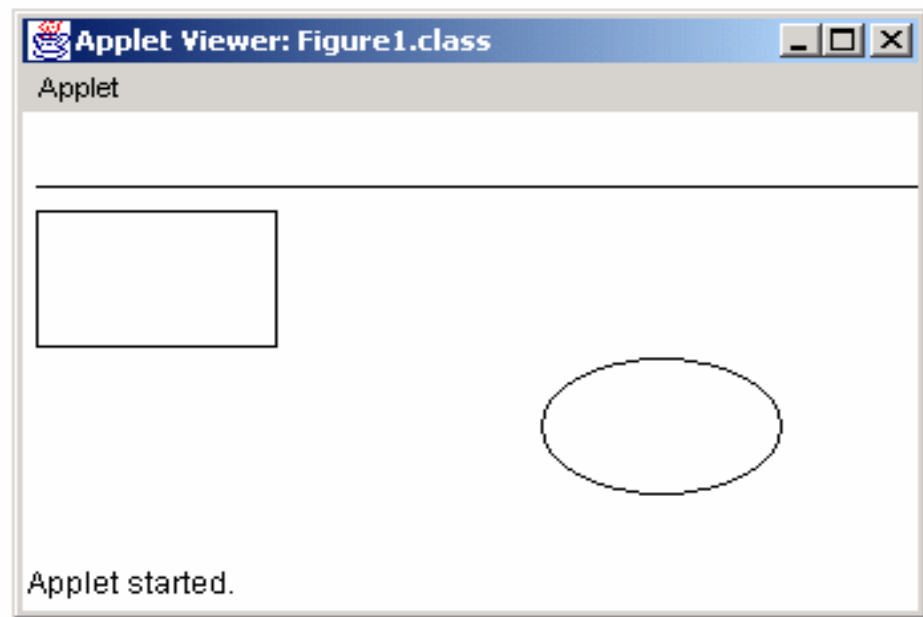


Check Your Progress 3

1)

```
// Drawing lines, rectangles and ovals.
import java.awt.*;
import java.applet.Applet;
    public class LinesRectsOvals extends Applet {
        // display various lines, rectangles and ovals
        public void paint( Graphics g )
        {
            g.drawLine( 5, 30, 350, 30 );
            g.drawRect( 5, 40, 90, 55 );
            g.drawOval( 195, 100, 90, 55 );
        } // end method paint
    }
```

Output:



2)

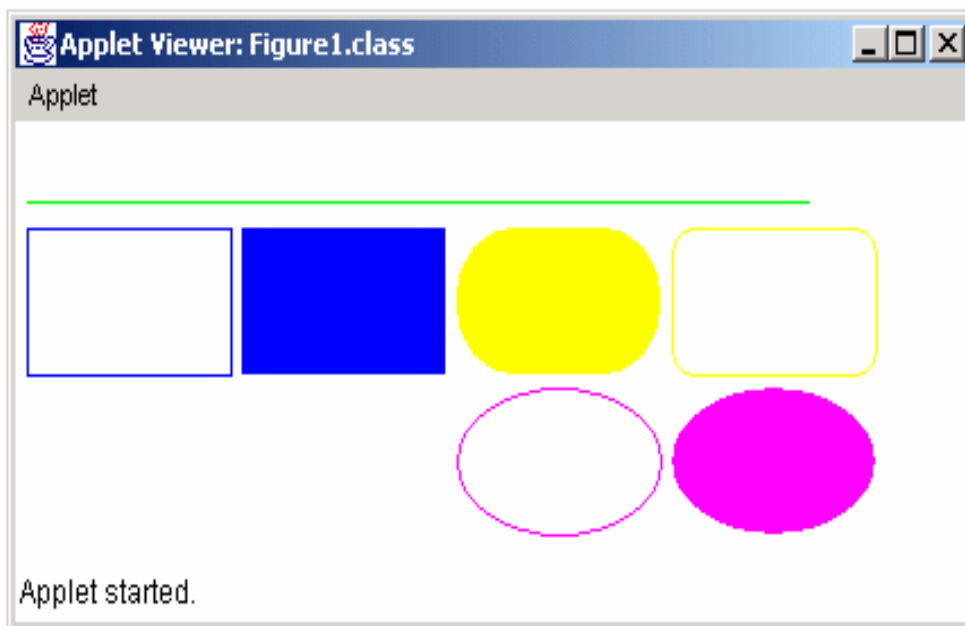
```
import java.awt.*;
import javax.swing.*;
    public class LinesRectsOvals extends JFrame
    {
        public void paint( Graphics g )
        {
            g.setColor( Color.GREEN );
            g.drawLine( 5, 30, 350, 30 );
            g.setColor( Color.BLUE );
            g.drawRect( 5, 40, 90, 55 );
        }
    }
```

```

        g.fillRect( 100, 40, 90, 55 );
//below two lines will draw a filled rounded rectangle with color yellow
        g.setColor( Color.YELLOW );
        g.fillRoundRect( 195, 40, 90, 55, 50, 50 );
//below two lines will draw a rounded figure of rectangle with color Pink
        g.drawRoundRect( 290, 40, 90, 55, 20, 20 );
        g.setColor( Color.PINK);
//below lines will draw an Oval figure with color Magenta
        g.setColor(Color.MAGENTA);
        g.drawOval( 195, 100, 90, 55 );
        g.fillOval( 290, 100, 90, 55 );
    }
}

```

Output:



3)

```

import java.awt.*;
class ChkGroup extends java.applet.Applet
{
    CheckboxGroup cbgr = new CheckboxGroup();
    Panel panel=new Panel();
    Frame fm=new Frame();
    Checkbox c1 = new Checkbox ("America Online");
    Checkbox c2 = new Checkbox ("MSN");
    Checkbox c3 = new Checkbox ("NetZero", cbgr, false);
    Checkbox c4 = new Checkbox ("EarthLink", cbgr, false);
    Checkbox c5 = new Checkbox ("Bellsouth DSL", cbgr, false);
    public void init()
    {
        fm.setVisible(true);
        fm.setSize(300,400);
        fm.add(panel);
        panel.add(c1);
        panel.add(c2);
        panel.add(c3);
        panel.add(c4);
        panel.add(c5);
    }
}

```

```

public static void main(String args[])
{
    ChkGroup ch=new ChkGroup();
    ch.init();
}
}

```

- American Online ☐ MSN ☐ NetZone ☐ EarthLink ☐ BellSouth DSL

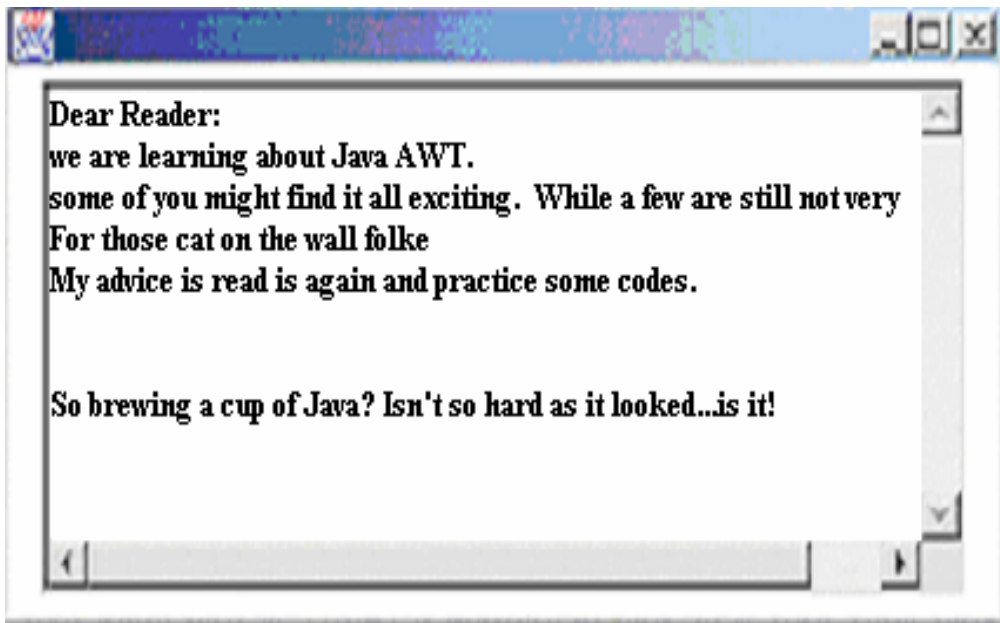
In the above examples, you will notice that the control which is not in the group has become a checkbox and the control which is in the group has become a radio button because checkbox allows you multiple selection but a radio button allows you a single selection at a time.

4)

```

import java.awt.*;
class TextFieldTest extends java.applet.Applet
{
    Frame fm=new Frame();
    Panel panel=new Panel();
    String letter = "Dear Readers: \n" +
    "We are learning about Java AWT. \n" +
    "Some of you might find it all exciting, while a few are
    still not very sure \n" +
    "For those cat on the wall folks \n" +
    "My advice is read it again and practice some codes. \n
    \n" +
    "So brewing a cup of Java? Isn't so hard as it looked...is
    it! " ;
    TextArea ltArea;
    public void init(){
        ltArea = new TextArea(letter, 10, 50);
        fm.setVisible(true);
        fm.setSize(300,400);
        fm.add(panel);
        panel.add(ltArea);
    }
    public static void main(String args[])
    {
        TextFieldTest tt=new TextFieldTest();
        tt.init();
    }
}

```



Basically, you will notice that along with the text area control you will see the scrollbars.

- 5) Swing components that subclass JComponent have many features, including: a pluggable look and feel that can be used to customize the look and feel when the program executes on different platforms.

We can have Shortcut keys (called mnemonics) for direct access to GUI components through the keyboard. It is very common event handling capabilities for cases where several GUI components initiate the same actions in the program. It gives the brief description of a GUI component's means (tool tips) that are displayed when the mouse cursor is positioned over the component for a short time. It has a support for technologies such as a Braille screen for blind people. It has support for user interface localization - customizing the user interface for display in different languages and cultural conventions.

Check Your Progress 4

- 1) A LayoutManager rearranges the components in the container based on their size, relative to the size of the container.

Consider the window that just popped up. It has five buttons of varying sizes. Resize the window, and watch how the buttons move. In particular try making it just wide enough, so that all the buttons fit on one line. Then, try making it narrow and tall, so that there is only one button on line. See if you can manage to cover up some of the buttons. Then uncover them. Note that whatever you try to do, the order of the buttons is maintained in a logical way. Button 1 is always before button 2, which is always before button 3, and so on.

It is harder to show, but imagine if the components changed sizes, as they might if you viewed this page in different browsers or on different platforms with different fonts.

The layout manager handles all these different cases for you to the greatest extent possible. If you had used absolute positioning, and the window was smaller than expected or the components larger than you expected, some components would likely be truncated or completely hidden. In essence, a layout manager defers decisions about positioning until runtime.

- 2) The repaint() method will cause AWT to invoke a component's update() method. AWT passes a Graphics object to update() –the same one that it passes to paint(). So, repaint() calls update() which calls paint().

- 3) BorderLayout
- 4) Stretchable: Button, Label and TextField
Non-Stretchable: CheckBox
- 5) java.awt.event
- 6) MouseListener and MouseMotionListener