**GUI BASICS**

When Java was introduced, the GUI classes are bundled in a library known as Abstract Windows Toolkit (AWT). The AWT contains several classes and methods that allow creating and managing windows.

GUI is a mediator between end user and the program.

**Console** is a program, provided by local OS which supports only text output.

In order to display GUI output we have another program similar to console provided by the local OS. This program is **Window**.

**The theory behind the GUI is as follows:**

The functionality of the constructor of the class, where GUI is to be created, is to create that particular GUI. **Strictly speaking, this is not the functionality of the constructor.** In fact, the constructor has some statements which are nothing but a call to a specific file in the local OS. This particular file will be responsible in creating the corresponding GUI. In the foreground it seems as if the GUI is created just because of the constructor. Whatever language we use to create the GUI (e.g. java, HTML, etc.) in all these cases, only the files present in the local OS will be responsible for creating the corresponding GUI. Because of this, the GUI created in all these cases will look alike.

When we want to create a GUI, we need to create object of that class which represent the corresponding GUI. **That means, we are making JVM to execute the files present in the local OS.** But remember that JVM can only execute bytecode, which is generated from .class file and this .class file is generated from .java file. The files present in the local operating system need not be in java and definitely they will not be in the java. They will be in some other native language.

**How can JVM execute the files present in the local OS to create the GUI?**

JVMdepends upon some classes known as **peer classes** to execute the files present in the local OS.

Thus, every AWT component depends upon the files of the local OS to create GUI, making JVM to execute these files which are non – java files. **This is reason why we call AWT classes (components) as heavy – weight components**.

From the above discussion, it is clear that, each OS will have its own files for creating GUI. If we create a GUI using java on one OS, then the look and feel of this GUI will be completely different when we execute the same java program on another OS.

**Java is not purely platform independent. Justify.**

Because of the dependency of AWT programs on the local OS in creating GUI, the output of these programs vary from one OS to another OS. The output of AWT programs are platform dependent. This makes java to be platform dependent. That is the reason, we say that Java is not purely platform independent.

**Disadvantage of AWT**

1. The AWT programs increase the overhead on the JVM.
2. AWT make Java to be platform dependent.

The above dis – advantages are overcome in **Swings**. Swing is an extension of AWT. The Swing components are not dependent upon the files of the local OS in creating the GUI and thereby, they decrease the overhead on the JVM. Thus, Swings are known as **light – weight components.**

**The Java GUI API**

The GUI API contains classes that can be classified into three groups:

1. Component class
2. Container class
3. Helper class

**Component classes**

An instance of component can be displayed on the screen. **Component** is the root class of all the user – interface classes including container class, and **JComponent** is the root class of all the Swing components. Both **Component** and **JComponent** are abstract classes.

For example, JButton, JLabel, JTextField etc..

**Container classes**

An instance of **Container** can hold instance of **Component**. Container classes are GUI components that are used to contain other GUI components. Window, Panel, Applet, Frame and Dialog are the container classes for AWT components. To work with Swing components, use Container, JFrame, JDialog, JApplet and JPanel.

|  |  |
| --- | --- |
| **Container classes** | **Description** |
| java.awt.Container | Is used to group components. Frame, Panel and Applet are its subclasses |
| javax.swing.JFrame | Is a window, not contained inside another window. It is used to hold other Swing user – interface components in Java GUI applications. |
| javax.Swing.Panel | Is an an invisible container that holds user – interface components. Panels can be nested. JPanel is often used as a canvas to draw graphics. |
| javax.Swing.JApplet | Is a subclass of Applet. You must extend JApplet to create a Swing based Java applet. |
| javax.Swing.JDialog | Is a pop – up window or message box generally used as a temporary window to receive additional information to the user. |

**Helper classes**

The helper classes, such as Graphics, Color, Font, FontMetrics, Dimension and LayoutManager, are not sub – classes of Component. They are used todescribe the properties of GUI components.

|  |  |
| --- | --- |
| **Helper classes** | **Description** |
| java.awt.Graphics | is an abstract class that provides the methods for drawing strings, lines, and simple shapes. |
| java.awt.Color | Deals with the colors of GUI components. |
| java.awt.Font | Specifies fonts for the text and drawings on GUI components. |
| java.awt.FontMetrics | Is an abstract class used to get the properties of Font. |
| java.awt.Dimension | Encapsulates the width and height of the component in a single object. |
| java.awt.LayoutManager | Specifies how components are arranged in a container. |

**Note:** The Helper classes are in **java.awt** package. The Swing component does not replace all the classes in AWT, only the AWT GUI component classes are replaced in Swing component.

**Frames**

To create a user interface, you need to create either a frame or an applet to hold the user interface components.

**Creating a Frame**

To create a Frame, use the JFrame class.

**// Program to create a Frame**

import javax.swing.JFrame;

class MyFrame

{

public static void main(String[] args)

{

JFrame frame = new JFrame("MyFrame");

frame.setSize(600, 200);

// frame.setLocation(400, 600);

frame.setLocationRelativeTo(null);

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

frame.setVisible(true);

}

}

If **setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE)** will not be invoked, then the console will not terminated after click on close button of frame. In that case press **Ctrl+C** to close the terminate batch job.

Till **setVisible(true)** is not invoked, then the frame will not visible. By default **setVisible** method is **false**.

**Methods defined in JFrame**

|  |  |
| --- | --- |
| **Methods** | **Description** |
| JFrame() | Creates a default frame with no title. |
| JFrame(String title) | Creates a frame with a specified title. |
| void setSize(int height, int width) | Sets the size of the frame |
| void setLocation(int x, int y) | Sets the upper left corner location of the frame |
| void setVisible(boolean visible) | Sets true to display the frame |
| void setDefaultCloseOperation(int mode) | Specifies the operation when the frame is closed. |
| void setLocationRelativeTo(Component c) | Sets the location of the frame relative to the specified component. If the component is null, the frame is centered on the screen. |
| void pack() | Automatically sets the frame size to build the components in the frame. |

JFrame is a top – level container to hold GUI components.

**NOTE: setSize(w,h)** should be invoked before **setLocationRelativeTo(),** otherwise the frame will not set to the center of the screen.

**Adding Components to Frame**

import javax.swing.\*;

class AddComponentFrame extends JFrame

{

JButton btn;

AddComponentFrame()

{

btn = new JButton("Button1");

setVisible(true);

setSize(200, 200);

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public static void main(String[] args)

{

AddComponentFrame adf = new AddComponentFrame();

}

}

In the above program, we have created the object of JButton component. But when we execute this program, the button will not displayed i.e. the button will not be added to the frame.

To add any component to a frame, **add(Component c)** method should be called by passing component object as parameter.

import javax.swing.\*;

class AddComponentFrame extends JFrame

{

AddComponentFrame()

{

JButton btn = new JButton("Button1");

add(btn);

setVisible(true);

setSize(200, 200); setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public static void main(String[] args)

{

AddComponentFrame adf = new AddComponentFrame();

}

}

Each **JFrame** contains a content pane. A content pane is an instance of java.awt.Container. The GUI components (such as Button) are placed in the content pane in a frame. In the earlier versions of Java, **getContentPane()** method had to be used in JFrame class to return the content pane of the Frame, then invoke the content pane’s **add()** method to place a component in the content pane.

import java.awt.\*;

import javax.swing.\*;

class AddComponentFrame extends JFrame

{

public static void main(String[] args)

{

JFrame frame = new JFrame();

JButton btn = new JButton("Button1");

**Container container = frame.getContentPane();**

**container.add(btn);**

frame.setVisible(true);

frame.setSize(200, 200);

frame.setLocationRelativeTo(null);

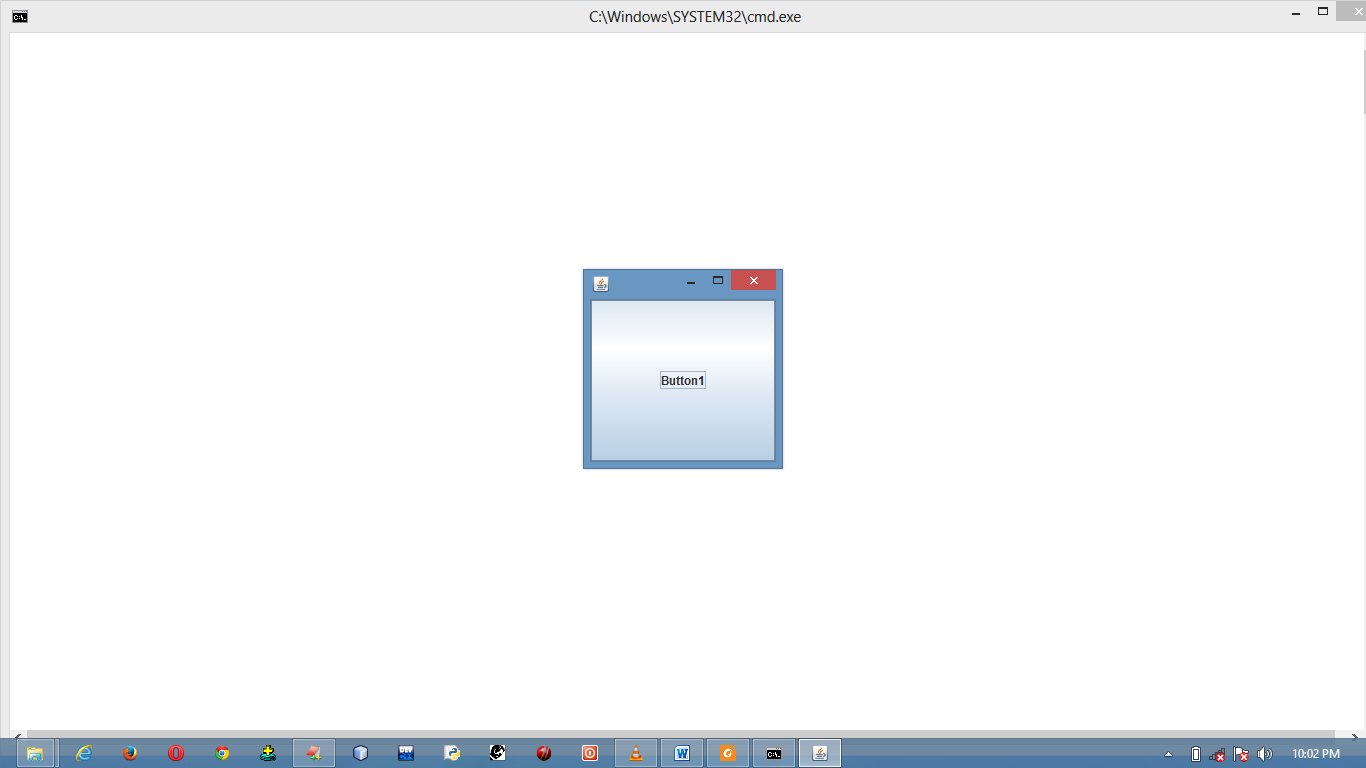
frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

}

But from Java5 to place components into content pane, just invoking **add()** method is required. This new feature is called **content – pane delegation.**

When we execute the program, following window will get displayed:



The button is always centered in the frame and occupies the entire frame.

**Let’s take another example**

import javax.swing.\*;

class AddComponentFrame extends JFrame

{

JButton jbtn1, jbtn2;

AddComponentFrame()

{

jbtn1 = new JButton("Button1");

jbtn2 = new JButton("Button2");

add(jbtn1);

add(jbtn2);

setSize(200, 200);

setLocationRelativeTo(null);

setVisible(true);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public static void main(String[] args)

{

new AddComponentFrame();

}

}

When we execute the above program, we will see only **Button2** on the window. This is because all the components have been added to the window, but one upon another. The last component added is **Button2**, that’s why only **Button2** is displayed.

This is because components are put into the frame by the content pane’s layout manager, and the default layout manager for the component pane places the button in the center.

So to make all the components visible, we have to align them properly on the window. We have some standard procedure, using which we can align the components in the frame.

The standard procedure which exactly specify the way in which we align the components on the container are known as **layout manager.**

A **layout manager** is an object that controls the size and position of components inside a Container object.

**Layout** is the boundary of a component on container.

**SUN defined 5 in – built layouts:**

1. Border Layout
2. Flow Layout
3. Grid Layout
4. GridBag layout
5. Card Layout

**Border Layout**

The border layout manager divides a container into five parts: East, West, South, North and Center as show below:

|  |  |  |
| --- | --- | --- |
| NORTH | | |
| WEST | CENTER | EAST |
| SOUTH | | |

The **BorderLayout** class has some **public static final String** constants to refer to the areas divided on the container according to border layout.

These constants are:

1. BorderLayout.EAST
2. BorderLayout.SOUTH
3. BorderLayout.WEST
4. BorderLayout.NORTH
5. BorderLayout.CENTER

The default layout associated with the Frame class object is BorderLayout. The default location associated with the BorderLayout is CENTER.

Therefore, when we call **add()** method on the frame class object, it takes layout to be **BorderLayout** and adds the component to the **CENTER** of it.

At each border only one component is visible. If we add multiple components at the same border, only the last added component is visible.

An important property of the **BorderLayout** is when a component is added to the **CENTER**, then the component not only occupies the **CENTER**, but also stretches to the other four areas, if there are no components in those areas.

Constructors used for BorderLayout manager:

1. public BorderLayout()
2. public BorderLayout(int hgap, int vgap) : creates BorderLayout with specified number of horizontal and vertical gap. Default gap is 0(zero).

**Program to demonstrate BorderLayout**

import javax.swing.\*;

import java.awt.\*;

class BorderLayoutDemo extends JFrame

{

JButton jbtn1, jbtn2, jbtn3, jbtn4, jbtn5;

BorderLayoutDemo()

{

jbtn1 = new JButton("East");

jbtn2 = new JButton("West");

jbtn3 = new JButton("North");

jbtn4 = new JButton("South");

jbtn5 = new JButton("Center");

setLayout(new BorderLayout());

add(jbtn1, BorderLayout.EAST);

add(jbtn2, BorderLayout.WEST);

add(jbtn3, BorderLayout.NORTH);

add(jbtn4, BorderLayout.SOUTH);

add(jbtn5, BorderLayout.CENTER);

setSize(300, 300);

setLocationRelativeTo(null);

setVisible(true);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public static void main(String[] args)

{

new BorderLayoutDemo();

}

}

**Flow Layout**

A FlowLayout arranges components in a directional flow. The components will be divided into rows and columns depending upon the components and their size. The flow direction is determined by the container’s **ComponentOrientation** property and may be one of the two values:

1. ComponentOrientation.LEFT\_TO\_RIGHT
2. ComponentOrientation.RIGHT\_TO\_LEFT

When we start adding components to the frame according to the FlowLayout, then the components would be added row – wise i.e. 1st row 🡪 col1, col2,.. Etc. If the row is filled, then the component would be added to 1st column of the 2nd row and so on.

The default alignment of this layout is centered layout. It means components are first added at first center row.

The **FlowLayout** class has some **public static final int** constants to refer to alignment on the container according to flow layout.

These constants are:

1. FlowLayout.LEFT
2. FlowLayout.RIGHT
3. FlowLayout.CENTER
4. FlowLayout.LEADING
5. FlowLayout.TRAILING

This class has below constructors to create its object:

1. public FlowLayout()
2. public FlowLayout(in align)
3. public FlowLayout(int align, int hgap, int vgap).

**FlowLayout is the default layout of Panel and Applet.**

import javax.swing.\*;

import java.awt.\*;

class FlowLayoutDemo extends JFrame

{

JButton jbtn1, jbtn2, jbtn3, jbtn4, jbtn5;

FlowLayoutDemo()

{

jbtn1 = new JButton("Button1");

jbtn2 = new JButton("Button2");

jbtn3 = new JButton("Button3");

jbtn4 = new JButton("Button4");

jbtn5 = new JButton("Button5");

setLayout(new FlowLayout(FlowLayout.RIGHT));

add(jbtn1);

add(jbtn2);

add(jbtn3);

add(jbtn4);

add(jbtn5);

setSize(300, 300);

setLocationRelativeTo(null);

setVisible(true);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public static void main(String[] args)

{

new FlowLayoutDemo();

}

}

We can also pass parameter **FlowLayout.LEFT** in add function as add(jbtn1, FlowLayout.LEFT). It will shifts the components to the left.

**Grid Layout**

It lays out a container’s component in a rectangular grid. The container is divided into equal – sized rectangles, and one component is placed in each rectangle. Each division is called a cell or grid.

In the GridLayout, only one component can be added to a grid and the component occupies the entire grid.

The components will be added to the grid in a serial fashion and we cannot skip the grid in between i.e. a component will not be added to grid 3, unless grid 2 is filled provided grid 1 is already occupied.

This class has below three constructors to create its object:

1. **public GridLayout()**
2. **public GridLayout(int rows, int cols)** : Creates a grid layout with the specified number of rows and columns. All components in the layout are given equal size. One, but not both, of rows and columns can be zero, which means that any number of objects can be placed in a row or in a column. If both the rows and columns will be non – zero, then rows is the dominating parameter i.e. the number of rows will be fixed and the layout manager dynamically calculates the number of columns.
3. **public Gridlayout(int rows, int cols, int hgap, int vgap)** : This constructor throws **java.lang.illigalArgumentException** if the values of both row and columns are set to be zero.

**Note**

When we use the GridLayout, it is compulsory that all the grids should be occupied with the components. No grid should be left free.

import javax.swing.\*;

import java.awt.\*;

class GridLayoutDemo extends JFrame

{

JButton jbtn1, jbtn2, jbtn3, jbtn4, jbtn5;

GridLayoutDemo()

{

jbtn1 = new JButton("Button1");

jbtn2 = new JButton("Button2");

jbtn3 = new JButton("Button3");

jbtn4 = new JButton("Button4");

jbtn5 = new JButton("Button5");

setLayout(new GridLayout(2, 4));

add(jbtn1);

add(jbtn2);

add(jbtn3);

add(jbtn4);

add(jbtn5);

setSize(300, 300);

setLocationRelativeTo(null);

setVisible(true);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public static void main(String[] args)

{

new GridLayoutDemo();

}

}

**GridBag Layout**

The GridBag Layout is similar to the GridLayout, but with some additional features. On the GridBagLayout, we have the facility of adding the components to a specific part of the grid (by mentioning the co – ordinate on all the four sides).

We can stretch the components between the grids, either horizontally or vertically.

**Note**

Even though, GridBagLayout is advantageous, programmers do not prefer it because it increases the complexity of the program.

We have a layout by the name **BoxLayout** in the **Swings** package. We use this layout in order to achieve all that we do using GridBagLayout and with less complexity.

This class uses below constructor to creates its object:

1. public GridBagLayout()

**Card Layout**

It treats each component in the container as a card. The CardLayout specifications helps us to add the components to a frame w.r.t. z – axis i.e. one component behind the other. Only one component is visible at a time. The first component added to a CardLayout object is the visible component when the container is first displayed.

This class has below two constructors to create its object:

1. public CardLayout()
2. public CardLayout(int hgap, int vgap).

**How can we apply layout to container or how can we change default layout of the container?**

In **Container** class we have below method to change or set layout:

**public void setLayout(LayoutManager mgr)**

For example,

setLayout(new FlowLayout())

**Null Layout**

We can also pass **null** as an argument to the **setLayout()** method.

Strictly speaking, null layout is not a layout. When we pass **null** as an argument to the **setLayout()**, it nullifies all the standard layouts i.e. the default layout will not be set for the corresponding components.

Now we have flexibility to add the components according to our specification.

We have a function **setBounds()** in the **Component** class. The **setBounds()** is used to apply custom layout to the components.

**public void setBounds(int x, int y, nt width, int height)**

Here, x denotes horizontal gap

y denotes vertical gap

width and height denotes the dimension of the component.

**Program to demonstrate custom layout**

import javax.swing.\*;

import java.awt.\*;

class CustomLayout extends JFrame

{

JLabel jlbl1, jlbl2;

JTextField jtxt1, jtxt2;

JButton jbtn1;

CustomLayout()

{

jlbl1 = new JLabel("UserName");

jlbl2 = new JLabel("Password");

jtxt1 = new JTextField(10);

jtxt2 = new JTextField(10);

jbtn1 = new JButton("Submit");

setLayout(null);

add(jlbl1);

add(jtxt1);

add(jlbl2);

add(jtxt2);

add(jbtn1);

jlbl1.setBounds(80, 100, 80, 20);

jtxt1.setBounds(200, 100, 100, 20);

jlbl2.setBounds(80, 130, 80, 20);

jtxt2.setBounds(200, 130, 100, 20);

jbtn1.setBounds(140, 160, 100, 20);

setVisible(true);

setSize(400, 400);

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public static void main(String[] args)

{

new CustomLayout();

}

}

**Note**

We cannot reuse the GUI components once created in a container (Frame).

For example, in the GUI programs which we have written till now, we have been creating objects of the TextField class wherever necessary. Instead whether it is for a Username label or a Password label, the TextField is same. If we have a TextField already and if we need one more, then we cannot add the one created earlier. Instead, we must create another one and add it.

**Panel**

Panel is a component which has the properties of container. It does not have the property of the window. Since, panel has the property of the container; we can add any number of components to the panel and now the panel will be treated as a single component.

**Advantages of Panel**

1. If we want to have more than one component in a grid, then we make use of the panel (As we know that in a Grid, only one component can be added).
2. If we want to have an empty grid, then we can add an empty panel (As we know that a Grid cannot be empty). This makes the grid look empty.

Panels act as a sub – container to group user interface components. Panels can be placed inside a frame or inside another panel.

import javax.swing.\*;

import java.awt.\*;

class PanelDemo extends JFrame

{

PanelDemo()

{

JPanel jpnl1 = new JPanel(new GridLayout(4, 3));

for (int i = 1; i <= 9; i++)

{

jpnl1.add(new JButton("" + i));

}

jpnl1.add(new JButton("" + 0));

jpnl1.add(new JButton("Start"));

jpnl1.add(new JButton("Stop"));

JPanel jpnl2 = new JPanel(new BorderLayout());

JTextField jtxt1 = new JTextField("Text to be shown here");

jpnl2.add(jtxt1, BorderLayout.NORTH);

jpnl2.add(jpnl1);

add(jpnl2);

JButton jbtn1 = new JButton("Food To be Placed Here");

add(jbtn1, BorderLayout.WEST);

setVisible(true);

setSize(400, 300);

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

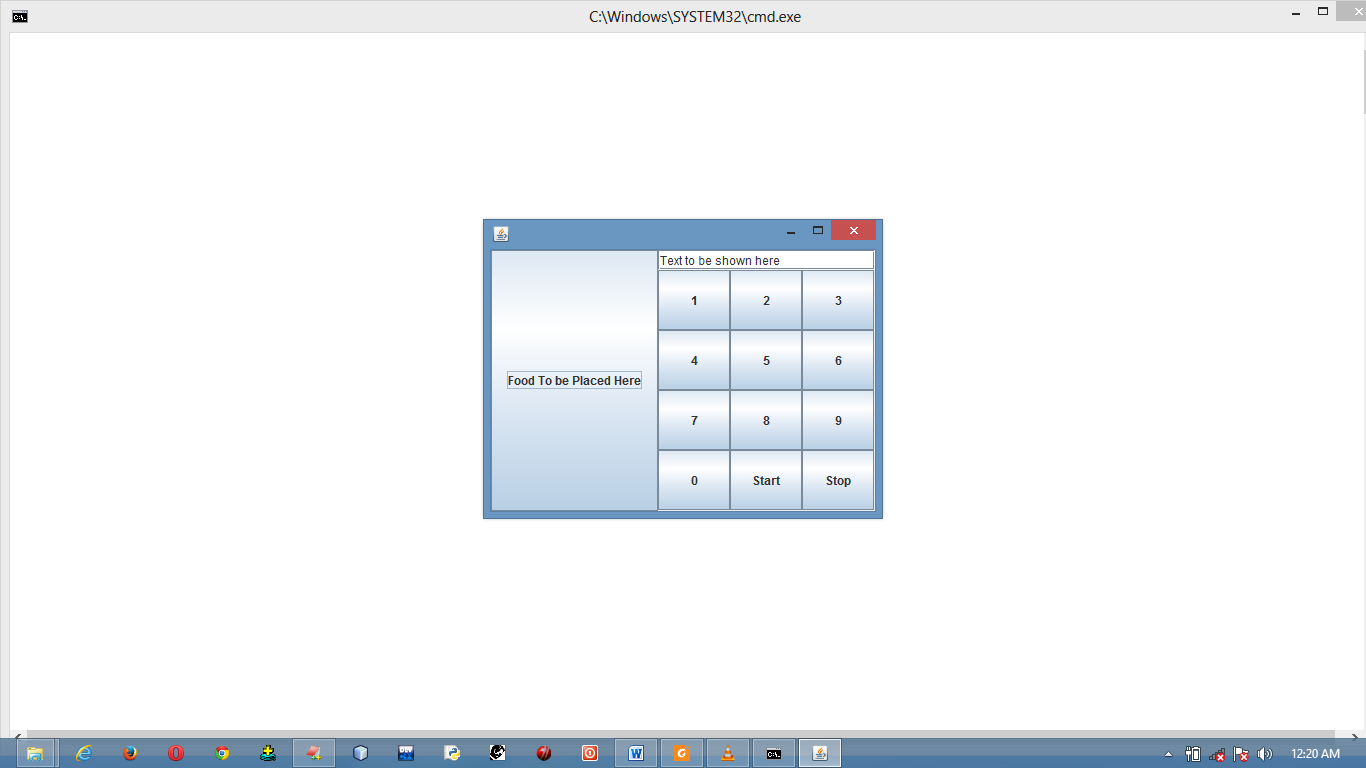
public static void main(String[] args)

{

new PanelDemo();

ss }

}

Output

**The Color class**

Colors for GUI components can be set by using the **java.awt.Color** class. Colors are made up from **Red**, **Blue** and **Green** components, each are represented by an **int** value that describes its intensity, ranging from o (darkest shade) to 255 (lightest shade). This is known as RGB model.

**Constructor to color class**

1. public color (int r, int g, int b)

The arguments **r**, **g** and **b** should be in between **0** to **255**, otherwise an **IllegalArgumentException** will occur.

**Methods used to set Color to the components**

1. setBackground (Color c)
2. setForeground (Color c)

**Program to demonstrate Color class**

import javax.swing.\*;

import java.awt.\*;

public class ColorDemo extends JFrame

{

ColorDemo()

{

JButton jbtn1 = new JButton("Red");

Color c1 = new Color(124, 234, 213);

Color c2 = new Color(234, 123, 230);

jbtn1.setBackground(c1);

jbtn1.setForeground(c2);

add(jbtn1);

setSize(100, 100);

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setVisible(true);

}

public static void main(String[] args)

{

new ColorDemo();

}

}

Alternatively we can use standard 13 colors (BLACK, BLUE, GREEN, CYAN, GRAY, DARK\_GRAY, LIGHT\_GRAY, ORANGE, YELLOW, RED, PINK, MAGENTA, WHITE) defined as constants in **java.awt.Color**.

import javax.swing.\*;

import java.awt.\*;

public class ColorDemo extends JFrame

{

ColorDemo()

{

JButton jbtn1 = new JButton("Red");

jbtn1.setBackground(Color.red);

jbtn1.setForeground(Color.green);

add(jbtn1);

setSize(100, 100);

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setVisible(true);

}

public static void main(String[] args)

{

new ColorDemo();

}

}

**The Font class**

We can create font using **java.awt.Font** class and set Fonts for the components using **setFont()** method in the **Component** class.

**Constructor to use Font class**

1. public Font (String s, int style, int size)

**Program to demonstrate Font class**

import javax.swing.\*;

import java.awt.\*;

class FontDemo extends JFrame

{

FontDemo()

{

JButton jbtn1 = new JButton("Font Test");

Font font = new Font("Times New Roman", Font.BOLD + Font.ITALIC, 16);

jbtn1.setFont(font);

add(jbtn1);

setVisible(true);

setSize(100, 100);

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public static void main(String[] args)

{

new FontDemo();

}

}

To get the Font names available in your system, you need to avail the instance of the class **java.awt.GraphicsEnvironment** class, using its static method **getLocalGraphicsEnvironment()** method. **GraphicsEnvironment** is an abstract class that describes a graphics environment on a particular system. There are two methods defined in this class:

1. String[] getAllFonts(): to obtain Font names available in the system.
2. Font[] getAvailableFontFamilyName(): to obtain all the available Fonts.

**Program to demonstrate getAvailableFontFamilyNames()**

import java.awt.GraphicsEnvironment;

class GraphicsEnvironmentDemo

{

public static void main(String[] args)

{

GraphicsEnvironment ge =

GraphicsEnvironment.getLocalGraphicsEnvironment();

String[] fontNames = ge.getAvailableFontFamilyNames();

for (int i = 0; i < fontNames.length; i++)

{

System.out.println(fontNames[i]);

}

}

}

**Program to demonstrate getAllFonts()**

import java.awt.GraphicsEnvironment;

import java.awt.Font;

class GraphicsEnvironmentDemo

{

public static void main(String[] args)

{

GraphicsEnvironment ge =

GraphicsEnvironment.getLocalGraphicsEnvironment();

Font[] fontNames = ge.getAllFonts();

for (int i = 0; i < fontNames.length; i++)

{

System.out.println(fontNames[i]);

}

}

}

**Common features of Swings GUI components**

**Tool Tip**: A tool tip is text displayed on the component when the mouse is being moved on the component. It is often used to describe the function of the component.

The **setToolTipText()** method is used to apply tool tip.

**Border:** We can set a border on any object of the component class using **setBorder(BorderType b)** method.

To create a titled border **new TitledBorder(String title)** method is used.

To create a line border **new LineBorder(Color color, int width)** method is used.

**Program demonstrating common swing GUI components**

import javax.swing.\*;

import java.awt.\*;

import javax.swing.border.\*;

class SwingComponents extends JFrame

{

JButton jLeft, jCenter, jRight;

JLabel jRed, jGreen;

SwingComponents()

{

jLeft = new JButton("Left");

jCenter = new JButton("Center");

jRight = new JButton("Right");

jRed = new JLabel("Red");

jGreen = new JLabel("Green");

JPanel p1 = new JPanel(new FlowLayout(FlowLayout.CENTER));

p1.add(jLeft);

p1.add(jCenter);

p1.add(jRight);

jRight.setToolTipText("This is Right Button");

p1.setBorder(new TitledBorder("Three Buttons"));

add(p1);

JPanel p2 = new JPanel(new GridLayout(1, 2));

p2.add(jRed);

p2.add(jGreen);

jRed.setForeground(Color.red);

jGreen.setForeground(Color.green);

p2.setBorder(new TitledBorder("Two Labels"));

jRed.setBorder(new LineBorder(Color.BLACK));

jGreen.setBorder(new LineBorder(Color.BLACK));

add(p2);

setVisible(true);

setSize(300, 200);

setLayout(new GridLayout(2, 1));

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public static void main(String[] args)

{

new SwingComponents();

}

}

**Note**

The same property may have different values for the different components. For example, the setVisible property is by default false for the JFrame, but it is true for the every instance of JComponent **(**JButton, JLabel etc.).

**Image Icons**

An icon is a fixed sized picture; typically it is small and used to decorate components. Images are normally stored in image files. Java currently supports three image formats:

1. GIF (Graphics Interchange Format)
2. JPEG (Joint Photographic Experts Group)
3. PNG (Portable Network Graphics)

The image file names for these types end with **.gif**, **.jpeg or .jpg** and **.png**

To display an image icon , create an **ImageIcon** object using **new java.swing.ImageIcon(fileName)**.

**Filenames are not case – sensitive in Windows but are in Unix**.

**Program to demonstrate Image Icons**

import javax.swing.\*;

class ImageIconDemo extends JFrame

{

ImageIconDemo()

{

JButton jBtn = new JButton(new ImageIcon("Java\_20110811.jpg"));

add(jBtn);

setVisible(true);

pack();

setLocationRelativeTo(null);

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

public static void main(String[] args)

{

new ImageIconDemo();

}

}