OPERATING SYSTEM (LAB)

Computer science Dep. / Fourth stage

DR. Mokhtar M. Hassan

MSC. Safa Sami

What is Java?

Java is a popular programming language created in 1995. It is owned by Oracle, and more than **3 billion** devices run Java. It is used for:

- Mobile applications (especially Android apps)
- Desktop applications
- Web applications
- Web servers and application servers
- Games
- Database connection
- And much, much more!

Why Use Java?

- Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.)
- It is one of the most popular programming languages in the world
- It has a large demand in the current job market
- It is easy to learn and simple to use
- It is open-source and free
- It is secure, fast and powerful
- It has huge community support (tens of millions of developers)
- Java is an object-oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs
- As Java is close to C++ and C#, it makes it easy for programmers to switch to Java or vice versa

Java Applications

- **Multithreaded** − With Java's multithreaded feature, it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that run smoothly.
- **Interpreted** − Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is incremental and lightweight.
- **High Performance** − With Just-In-Time compilers, Java enables high performance.
- **Distributed** − Java is designed for the distributed environment of the internet.
- **Dynamic** − Java is more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry an extensive amount of

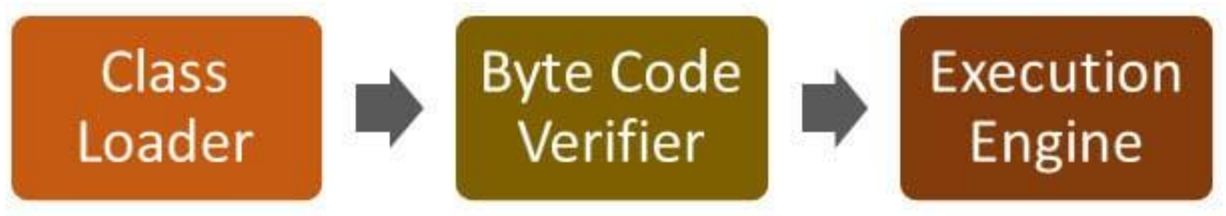
run-time information that can be used to verify and resolve accesses to objects on run-time.

What is JVM?

Java Virtual Machine (JVM) is an engine that provides a runtime environment to drive the Java Code or applications. It converts Java bytecode into machine language. JVM is a part of Java Runtime Environment (JRE). In other programming languages, the compiler produces machine code for a particular system. However, the Java compiler produces code for a Virtual Machine known as Java Virtual Machine.

How JVM Works?

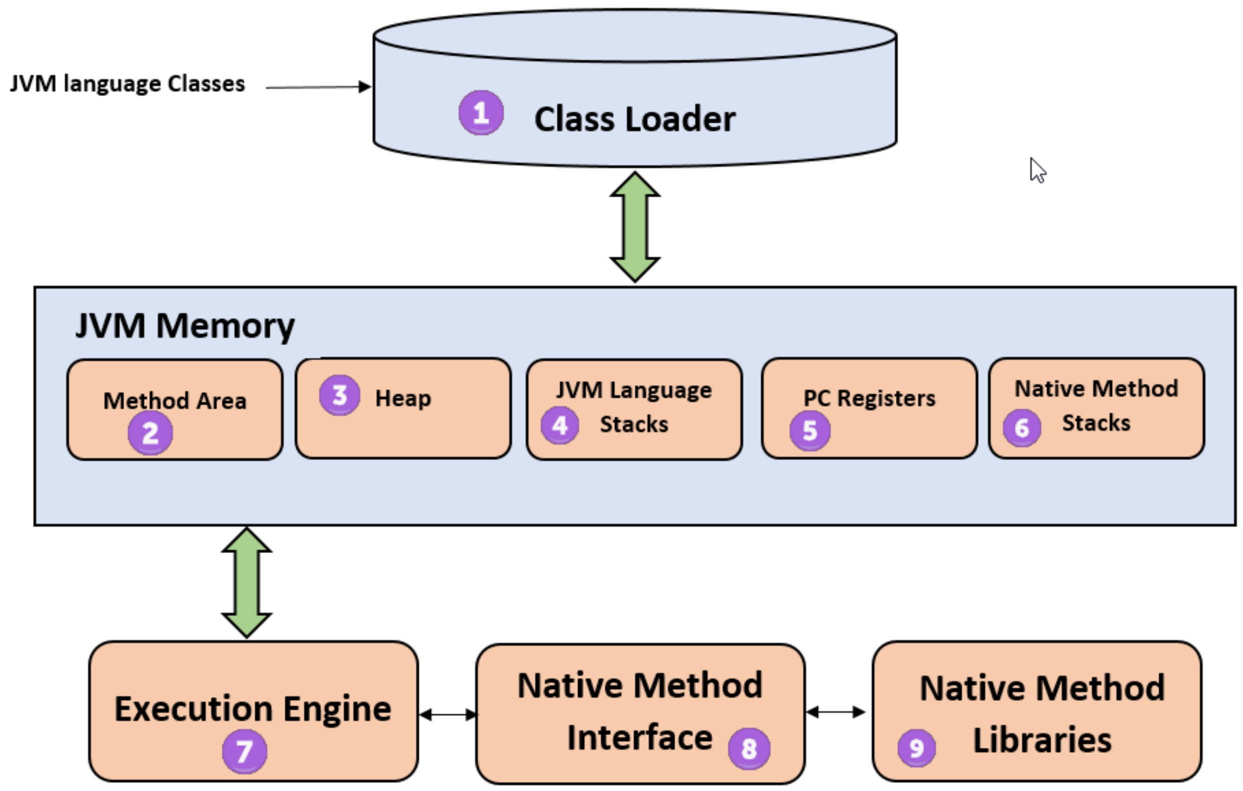
First, Java code is compiled into bytecode. This bytecode gets interpreted on different machines. Between the host system and Java source, Bytecode is an intermediary language. JVM in Java is responsible for allocating memory space.



Working of Java Virtual Machine (JVM)

JVM Architecture

Now in this JVM tutorial, let’s understand the Architecture of JVM. JVM architecture in Java contains a class loader, memory area, execution engine etc.



Java Virtual Machine Architecture

- **Class Loader**

The class loader is a subsystem used for loading class files. It performs three major functions viz. Loading, Linking, and Initialization.

- **Method Area**

JVM Method Area stores class structures like metadata, the constant runtime pool, and the code for methods.

- **Heap**

All the Objects, their related instance variables, and arrays are stored in a heap. This memory is common and shared across multiple threads.

- **JVM language Stacks**

Java language Stacks store local variables and their partial results. Each thread has its own JVM stack, created simultaneously as the thread is created. A new frame is created whenever a method is invoked, and it is deleted when the method invocation process is complete.

- **PC Registers**

PC register stores the address of the Java virtual machine instruction which is currently executing. In Java, each thread has its separate PC register.

- **Native Method Stacks**

Native method stacks hold the instruction of native code depending on the native library. It is written in another language instead of Java.

- **Execution Engine**

It is software used to test hardware, software, or complete systems. The test execution engine never carries any information about the tested product.

- **Native Method interface**

The Native Method Interface is a programming framework. It allows Java code running in a JVM to call by libraries and native applications.

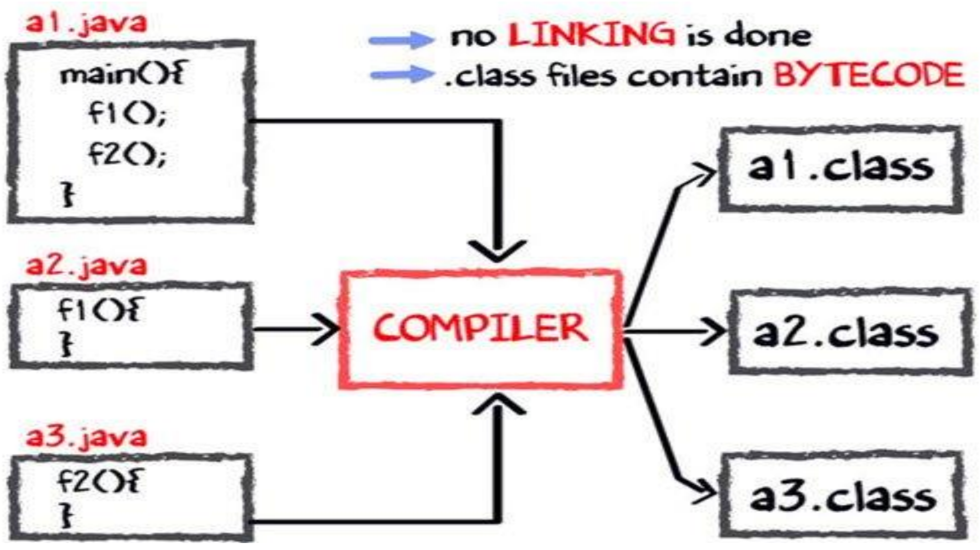
- **Native Method Libraries**

Native Library is a collection of the Native Libraries (C, C++) needed by the Execution Engine.

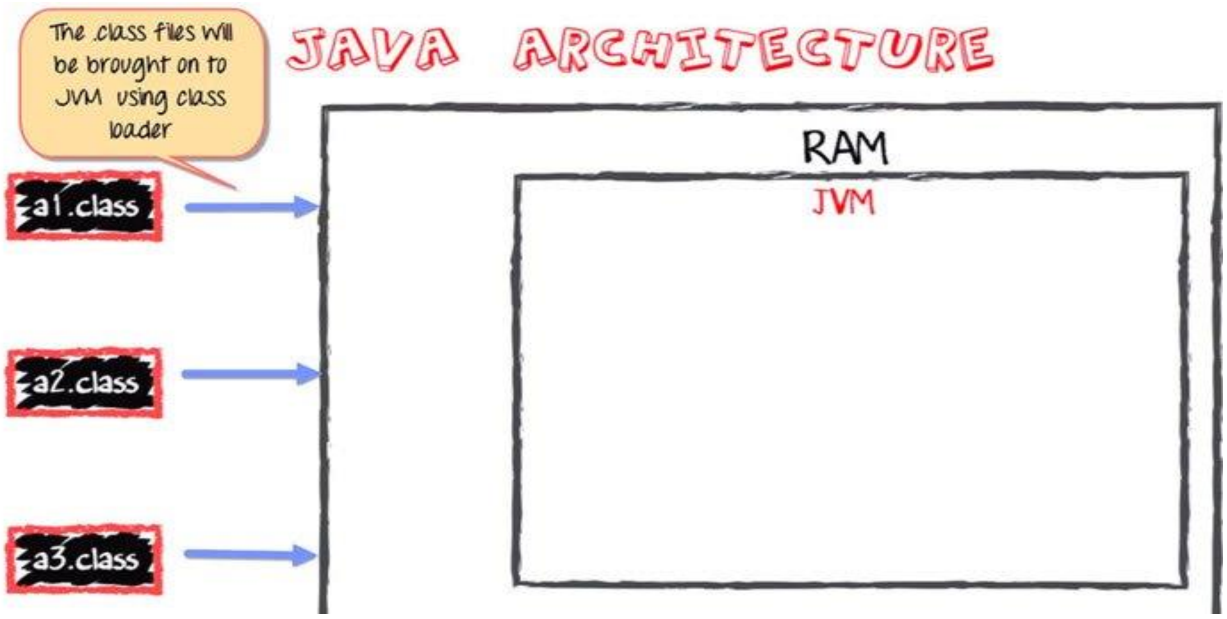
Java code Compilation and Execution in Java VM

Now in this JVM tutorial, let's look at the process for JAVA. In your main, you have two methods, f1 and f2.

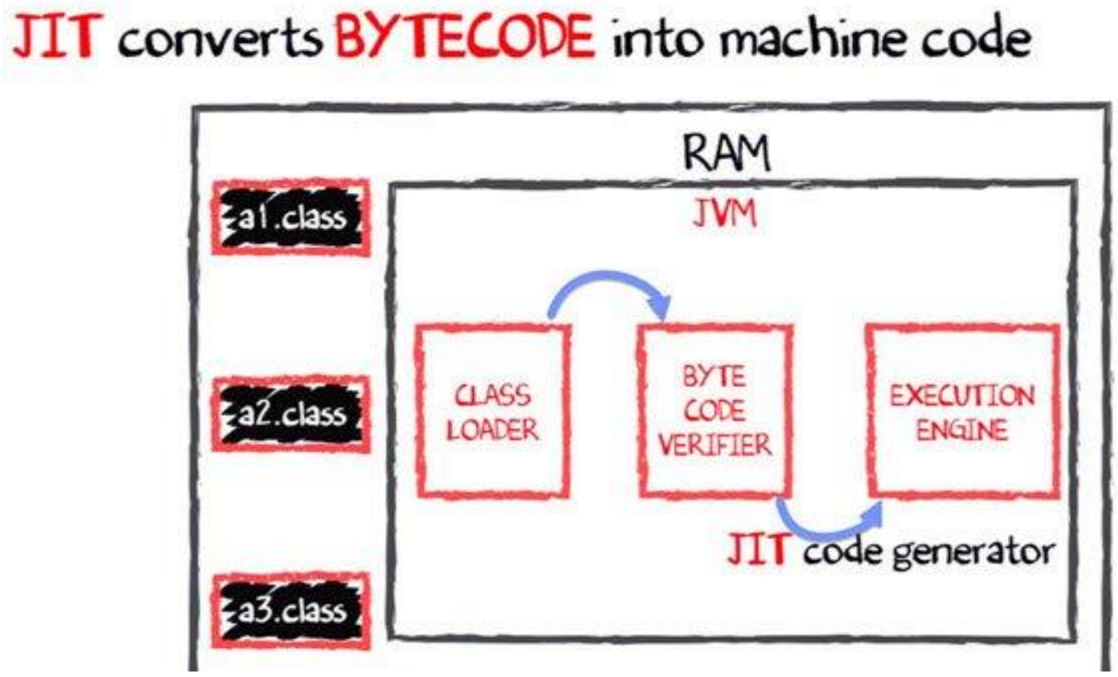
- The main method is stored in file a1.java
- f1 is stored in a file as a2.java
- f2 is stored in a file as a3.java



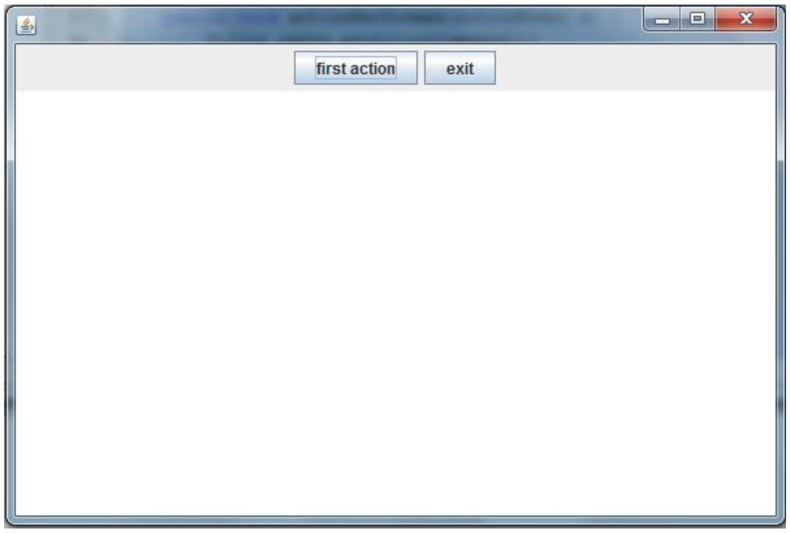
The compiler will compile the three files and produces 3 corresponding .class file, which consists of BYTE code. Unlike C, no linking is done.
The Java VM or Java Virtual Machine resides on the RAM. During execution, using the class loader, the class files are brought to the RAM. The BYTE code is verified for any security breaches.


The .class files will be brought on to JVM using class loader
JAVA ARCHITECTURE
RAM
JVM
a1.class
a2.class
a3.class

Next, the execution engine will convert the Bytecode into Native machine code. This is just in time compiling. It is one of the main reasons why Java is comparatively slow.



```
package javaapplication17;

import java.awt.Dimension;
import java.awt.Point;
import java.awt.Toolkit;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Container;

import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;

public class JavaApplication17 extends JFrame implements ActionListener {


    public JavaApplication17() {
        setTitle("My First Java Example with Netbeans");

        Dimension d=Toolkit.getDefaultToolkit().getScreenSize();
        setSize(d.width*4/5,d.height*4/5);



        Dimension d1=Toolkit.getDefaultToolkit().getScreenSize();
        Dimension d2=getSize();
        Point p=new Point(d1.width/2-d2.width/2, (int)d1.height/2-d2.height/2);
        setLocation(p);

        Container contents=getContentPane();



        setLayout( new FlowLayout() );

        JButton btn1 = new JButton("click me");
        contents.add(btn1);

        JButton btn2 = new JButton("exit");
        contents.add(btn2);
        btn1.addActionListener(this);
        btn2.addActionListener(this);



        //setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );


        setVisible(true);
    }

    public static void main(String[] args) {
     new JavaApplication17();
    }

     public void actionPerformed(ActionEvent e) {
        String s=e.getActionCommand();
        if(s=="click me")
            JOptionPane.showMessageDialog(null, "you clicked me");
        else
        {
            JOptionPane.showMessageDialog(null, "you have chosed to exit");
            System.exit(0);
        }


     }


 }
```



```
package javaapplication18;

import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextArea;


public class JavaApplication18 extends JFrame implements ActionListener{

   public JavaApplication18 () {
      int fwidth=600, fheight=400;



      Dimension d=Toolkit.getDefaultToolkit().getScreenSize();
      setLocation(d.width/2-fwidth/2,d.height/2-fheight/2);
      setSize(fwidth,fheight);
```

```
        Container cont=getContentPane();
        //setLayout(new BorderLayout());
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        panelBtns=new JPanel(new FlowLayout());
        cont.add(panelBtns, BorderLayout.PAGE_START);

        text1=new JTextArea();
        cont.add(text1, BorderLayout.CENTER);


        btn1=new JButton("first action");
        btn1.addActionListener(this);
        //cont.add(btn1, BorderLayout.PAGE_START);
        panelBtns.add(btn1);

        btn2=new JButton("exit");
        btn2.addActionListener(this);
        //cont.add(btn2, BorderLayout.PAGE_START);
        panelBtns.add(btn2);



        setVisible(true);
    }

    public static void main(String[] args) {
        JavaApplication18 obj=new JavaApplication18();
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        String cmd=e.getActionCommand();
        if(cmd.equals(btn1.getText())) {
            text1.append("i am clicked\n");
        }
          if(cmd.equals(btn2.getText())) {
            System.exit(0);
        }
       }
     JButton btn1, btn2;        JTextArea text1;        JPanel panelBtns;
 }
```

LayoutManagers:

The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes of layout managers. There are following classes that represents the layout managers:

1. java.awt.BorderLayout
2. java.awt.FlowLayout
3. java.awt.GridLayout
4. java.awt.CardLayout
5. java.awt.GridBagLayout
6. javax.swing.BoxLayout
7. javax.swing.GroupLayout
8. javax.swing.ScrollPaneLayout
9. javax.swing.SpringLayout etc.

BorderLayout:

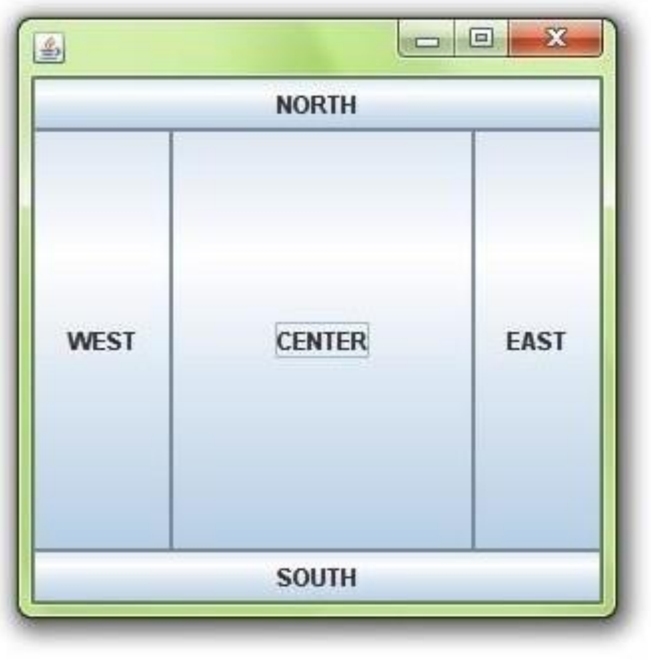
The BorderLayout is used to arrange the components in five regions: north, south, east, west and center. Each region (area) may contain one component only. It is the default layout of frame or window (such as dialog). The BorderLayout provides five constants for each region:

1. **public static final int NORTH**
2. **public static final int SOUTH**
3. **public static final int EAST**
4. **public static final int WEST**
5. **public static final int CENTER**

Constructors of BorderLayout class:

- **BorderLayout():** creates a border layout but with no gaps between the components.
- **JBorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components.

Example of BorderLayout class:



Method 1:

```
package example3;


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

public class Example3 {
JFrame f;
Example3(){
        f=new JFrame();

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

        f.add(b1,BorderLayout.NORTH);
        f.add(b2,BorderLayout.SOUTH);
        f.add(b3,BorderLayout.EAST);
        f.add(b4,BorderLayout.WEST);
        f.add(b5,BorderLayout.CENTER);
        f.setSize(300,300);
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
}
public static void main(String[] args) {
        Example3 ex=new Example3();
}
}
```

Method 2

```
package example3;


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

public class Example3 extends JFrame{

Example3(){

   Container f=getContentPane();

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

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



        setSize(300,300);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setVisible(true);
}
public static void main(String[] args) {
        Example3 ex=new Example3();
}
}
```

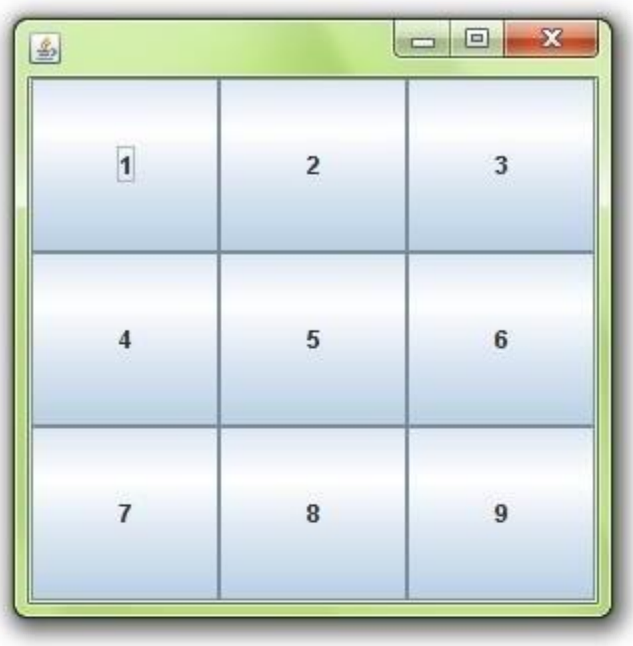
GridLayout

The GridLayout is used to arrange the components in rectangular grid. One component is displayed in each rectangle.

Constructors of GridLayout class:

1. **GridLayout():** creates a grid layout with one column per component in a row.
2. **GridLayout(int rows, int columns):** creates a grid layout with the given rows and columns but no gaps between the components.
3. **GridLayout(int rows, int columns, int hgap, int vgap):** creates a grid layout with the given rows and columns alongwith given horizontal and vertical gaps.

Example of GridLayout class:



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

```
public class MyGridLayout{
JFrame f;
MyGridLayout(){
   f=new JFrame();

   JButton b1=new JButton("1");
   JButton b2=new JButton("2");
   JButton b3=new JButton("3");
   JButton b4=new JButton("4");
   JButton b5=new JButton("5");
      JButton b6=new JButton("6");
      JButton b7=new JButton("7");
   JButton  b8=new  JButton("8");
      JButton b9=new JButton("9");

   f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
   f.add(b6);f.add(b7);f.add(b8);f.add(b9);

   f.setLayout(new GridLayout(3,3));
   //setting grid layout of 3 rows and 3 columns

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

FlowLayout

The FlowLayout is used to arrange the components in a line, one after another (in a flow). It is the default layout of applet or panel.

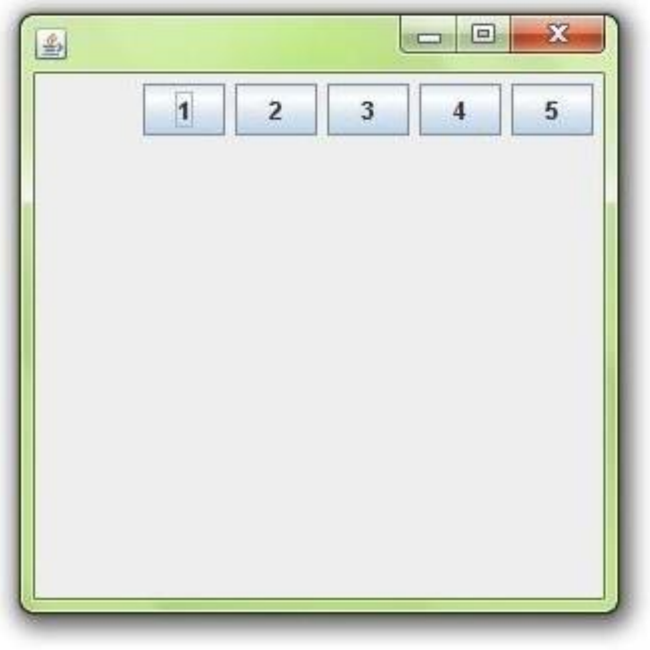
Fields of FlowLayout class:

1. **public static final int LEFT**
2. **public static final int RIGHT**
3. **public static final int CENTER**
4. **public static final int LEADING**
5. **public static final int TRAILING**

Constructors of FlowLayout class:

1. **FlowLayout():** creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.
2. **FlowLayout(int align):** creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
3. **FlowLayout(int align, int hgap, int vgap):** creates a flow layout with the given alignment and the given horizontal and vertical gap.

Example of FlowLayout class:



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

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

    JButton b1=new JButton("1");
    JButton b2=new JButton("2");
    JButton b3=new JButton("3");
    JButton b4=new JButton("4");
    JButton b5=new JButton("5");

    f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);

    f.setLayout(new FlowLayout(FlowLayout.RIGHT));
    //setting flow layout of right alignment

    f.setSize(300,300);
```

```
   f.setVisible(true);
}
public static void main(String[] args) {
   new MyFlowLayout();
}
}
```

BoxLayout class:

The BoxLayout is used to arrange the components either vertically or horizontally. For this purpose, BoxLayout provides four constants. They are as follows:

Note: BoxLayout class is found in javax.swing package.

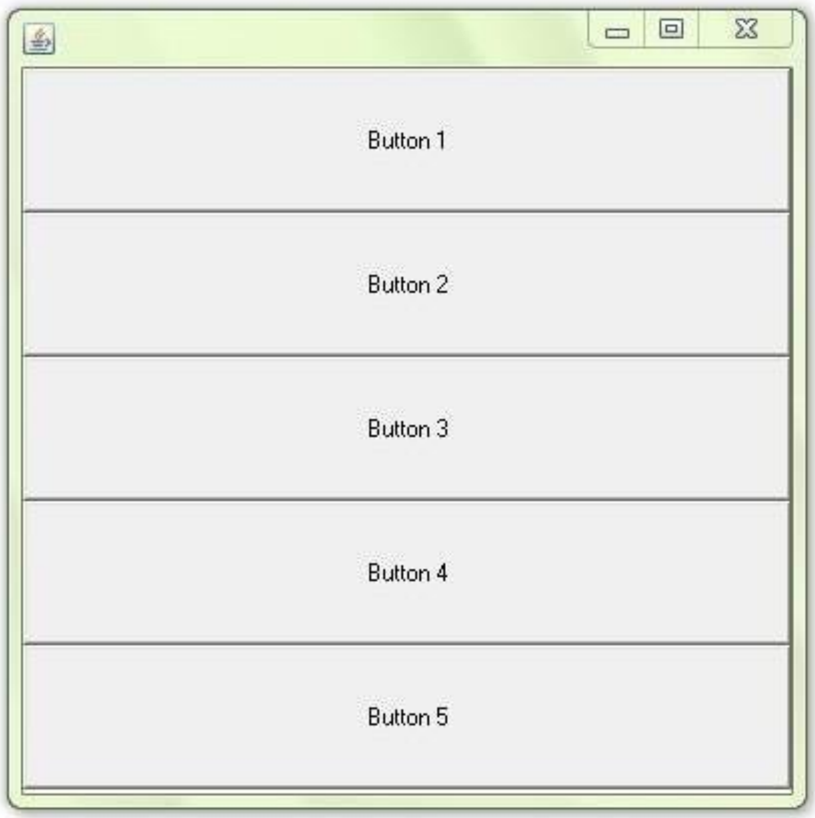
Fields of BoxLayout class:

1. **public static final int X_AXIS**
2. **public static final int Y_AXIS**
3. **public static final int LINE_AXIS**
4. **public static final int PAGE_AXIS**

Constructor of BoxLayout class:

1. **BoxLayout(Container c, int axis):** creates a box layout that arranges the components with the given axis.

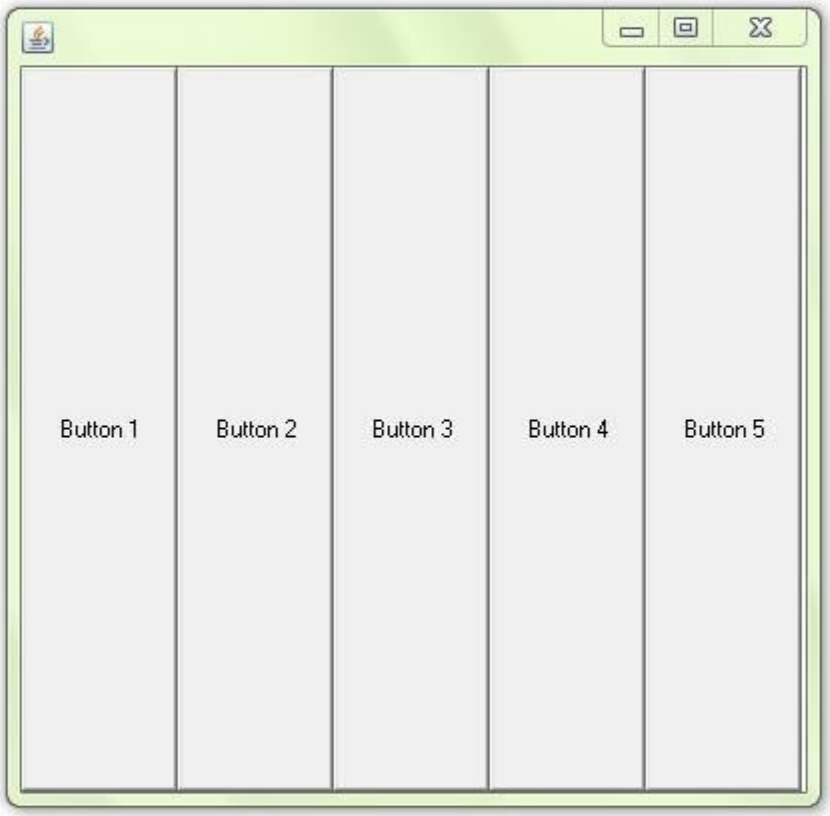
Example of BoxLayout class with Y-AXIS:



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

public class BoxLayoutExample1 extends Frame {
 Button buttons[];

 public BoxLayoutExample1 () {
   buttons = new Button [5];

   for (int i = 0;i<5;i++) {
      buttons[i] = new Button ("Button " + (i + 1));
      add (buttons[i]);
    }

setLayout (new BoxLayout (this, BoxLayout.Y_AXIS));
setSize(400,400);
setVisible(true);
}

public static void main(String args[]){
BoxLayoutExample1 b=new BoxLayoutExample1();
}
}
```

download this example

Example of BoxLayout class with X-AXIS:



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

public class BoxLayoutExample2 extends Frame {
 Button buttons[];
 public BoxLayoutExample2() {
  buttons = new Button [5];

  for (int i = 0;i<5;i++) {
     buttons[i] = new Button ("Button " + (i + 1));
     add (buttons[i]);
   }

setLayout (new BoxLayout(this, BoxLayout.X_AXIS));
setSize(400,400);
setVisible(true);
}

public static void main(String args[]){
BoxLayoutExample2 b=new BoxLayoutExample2();
}
}
```

CardLayout class

The CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout.

Constructors of CardLayout class:

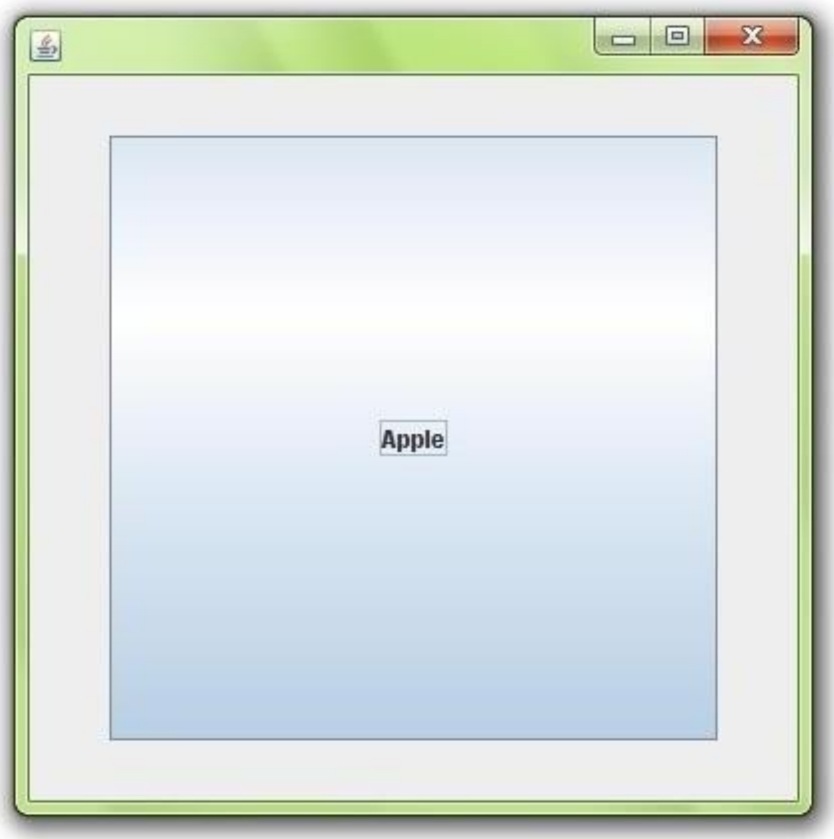
1. **CardLayout():** creates a card layout with zero horizontal and vertical gap.
2. **CardLayout(int hgap, int vgap):** creates a card layout with the given horizontal and vertical gap.

Commonly used methods of CardLayout class:

- **public void next(Container parent):** is used to flip to the next card of the given container.

- **public void previous(Container parent):** is used to flip to the previous card of the given container.
- **public void first(Container parent):** is used to flip to the first card of the given container.
- **public void last(Container parent):** is used to flip to the last card of the given container.
- **public void show(Container parent, String name):** is used to flip to the specified card with the given name.

Example of CardLayout class:



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

import javax.swing.*;
```

```java
public class CardLayoutExample extends JFrame implements ActionListener{
CardLayout card;
JButton b1,b2,b3;
Container c;
   CardLayoutExample(){

      c=getContentPane();
      card=new CardLayout(40,30);
//create CardLayout object with 40 hor space and 30 ver space
      c.setLayout(card);

      b1=new JButton("Apple");
      b2=new JButton("Boy");
      b3=new JButton("Cat");
      b1.addActionListener(this);
      b2.addActionListener(this);
      b3.addActionListener(this);

      c.add("a",b1);c.add("b",b2);c.add("c",b3);

   }
   public void actionPerformed(ActionEvent e) {
   card.next(c);
   }

   public static void main(String[] args) {
      CardLayoutExample cl=new CardLayoutExample();
      cl.setSize(400,400);
      cl.setVisible(true);
      cl.setDefaultCloseOperation(EXIT_ON_CLOSE);
   }
}
```

GridBagLayout

`GridBagLayout` is a very flexible layout manager that allows you to position components relative to one another using constraints. With `GridBagLayout` (and a fair amount of effort) you can create almost any imaginable layout.

Components are arranged at "logical" coordinates on a abstract grid. We'll call them "logical" coordinates because they really designate positions in the space of rows and columns formed by the set of components. Rows and columns of the grid stretch to different sizes, based on the sizes and constraints of the components they hold.

GridBagConstraints

The appearance of a grid bag layout is controlled by sets of GridBagConstraints, and that's where things get hairy. Each component managed by a GridBagLayout is associated with a GridBagConstraints object. GridBagConstraints holds the following variables, which we'll describe in detail shortly:

int gridx, gridy

Controls the position of the component on the layout's grid.

int weightx, weighty

Controls how additional space in the row or column is allotted to the component.

int fill

Controls whether the component expands to fill the space allocated to it.

int gridheight, gridwidth

Controls the number of rows or columns the component occupies.

int anchor

Controls the position of the component if there is extra room within the space allocated for it.

int ipadx, ipady

Controls padding between the component and the borders of it's area.

Insets insets

Controls padding between the component and neighboring components.

To make a set of constraints for a component or components, you simply create a new instance of GridBagConstraints and set these public variables to the appropriate values. There are no pretty constructors, and not much else to the class at all.

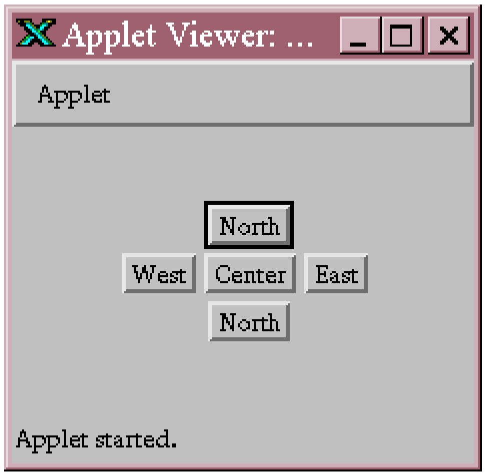
```
Component component = new Label("constrain me, please...");
GridBagConstraints constraints = new GridBagConstraints;
constraints.gridx = x;
constraints.gridy = y;
...
add( component, constraints );
```

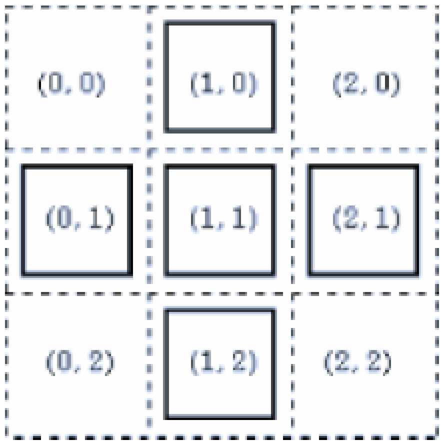
Grid Coordinates

One of the biggest surprises in the `GridBagLayout` is that there's no way to specify the size of the grid. There doesn't have to be. The grid size is determined implicitly by the constraints of all the objects; the layout manager picks dimensions large enough so that everything fits.

Thus, if you put one component in a layout and set its gridx and gridy constraints to 25, the layout manager creates a 25 x 25 grid, with rows and columns both numbered from 0 to 24. If you add a second component with a gridx of 30 and a gridy of 13, the grid's dimensions change to 30 x 25. You don't have to worry about setting up an appropriate number of rows and columns. The layout manager does it automatically, as you add components.

"A Simple Layout"





```
import java.awt.*;
public class GridBag1 extends java.applet.Applet {
    GridBagConstraints constraints = new GridBagConstraints();
    void addGB( Component component, int x, int y  ) {
        constraints.gridx = x;
        constraints.gridy = y;
        add ( component, constraints );
    }

    public void init() {
        setLayout( new GridBagLayout() );
        int x, y; // for clarity
        addGB(  new  Button("North"),  x=1,y=0  );
        addGB(  new  Button("West"),  x=0,y=1  );
        addGB( new Button("Center"), x=1,y=1 );
        addGB(  new  Button("East"),  x=2,y=1  );
        addGB( new Button("North"), x=1,y=2 );
    }
}
```

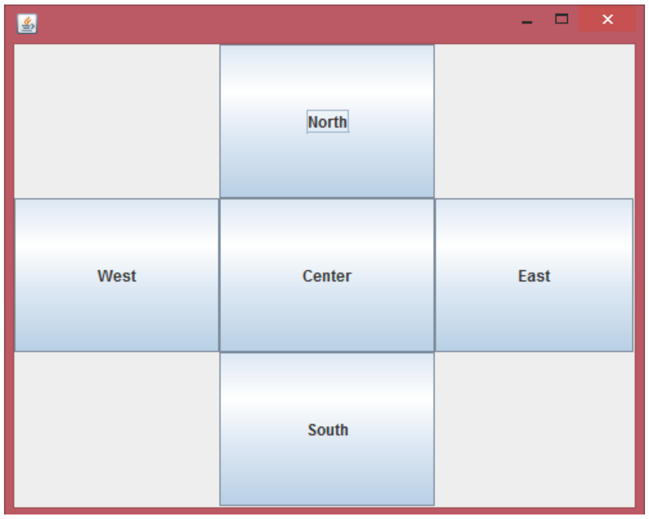
Or ,in another way:

```
package example3;


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

public class Example3 extends JFrame {

   public Example3() {
      setLayout( new GridBagLayout() );
     Container c =getContentPane();
```

```
        GridBagConstraints constraints = new GridBagConstraints();
        JButton b1=new JButton("North");
        constraints.gridx=1; constraints.gridy=0;
        c.add(b1, constraints);

        JButton b2=new JButton("West");
        constraints.gridx=0; constraints.gridy=1;
        c.add(b2, constraints);

        JButton b3=new JButton("Center");
        constraints.gridx=1; constraints.gridy=1;
        c.add(b3, constraints);

        JButton b4=new JButton("East");
        constraints.gridx=2; constraints.gridy=1;
        c.add(b4, constraints);

        JButton b5=new JButton("South");
        constraints.gridx=1; constraints.gridy=2;
        c.add(b5, constraints);

        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(500,400);
        setVisible(true);
    }

    public static void main(String[] args) {
        Example3 cl=new Example3();
    }
}
```

Fill

Now let's make the buttons expand to fill the entire applet. To do so, we must take two steps: we must set the `fill` constraint for each button to the value `BOTH`, and we must set the `weightx` and `weighty` values to the same nonzero value.



```
constraints.weightx = 1.0;
 constraints.weighty = 1.0;
 constraints.fill = GridBagConstraints.BOTH;
```

`BOTH` is one of the constants of the `GridBagConstraints` class; it tells the component to fill the available space in both directions. The following table lists the constants that you can use to set the `fill` field:

`HORIZONTAL`

Fill the available horizontal space.

`VERTICAL`

Fill the available vertical space.

`BOTH`

Fill the available space in both directions.

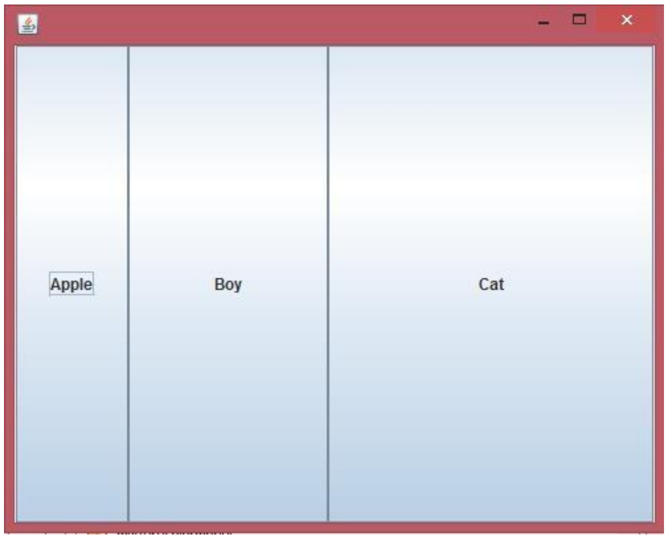
`NONE`

Don't fill the available space; display the component at its preferred size.

We set the weight constraints to 1.0; it doesn't matter what they are, provided that each component has the same non-zero weight. `fill` doesn't work if the component weights in the direction you're filling are 0, which is the default value.

Weighting

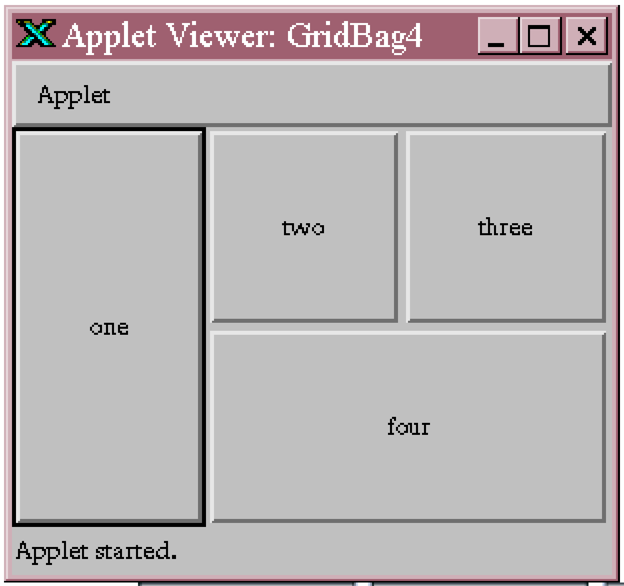
The `weightx` and `weighty` variables of a `GridBagConstraints` object determine how space is distributed among the columns or rows in a layout. As long as you keep things simple, the effect these variables have is fairly intuitive: the larger the weight, the greater the amount of space allocated to the component. The display below shows what happens if we vary the `weightx` constraint from 0.1 to 1.0 as we place three buttons in a row.

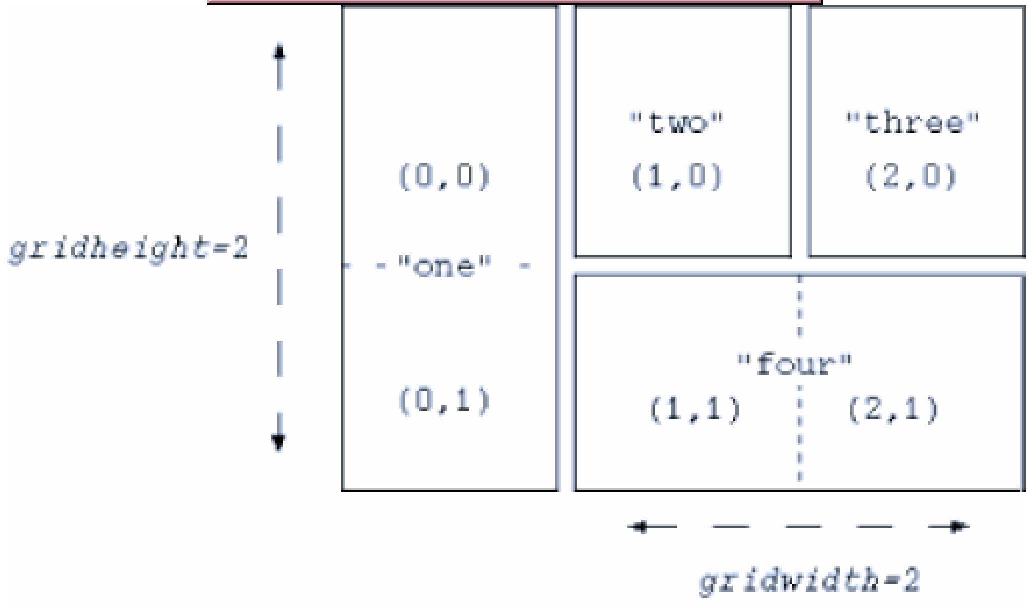


```
constraints.weighty=1;
constraints.fill = GridBagConstraints.BOTH;

constraints.weightx=0.1;
JButton b1=new JButton("Apple");
constraints.gridx=1; constraints.gridy=0;
c.add(b1, constraints);

constraints.weightx=0.5;
JButton b2=new JButton("Boy");
constraints.gridx=2; constraints.gridy=0;
c.add(b2, constraints);

constraints.weightx=1;
JButton b3=new JButton("Cat");
constraints.gridx=3; constraints.gridy=0;
c.add(b3, constraints);
```

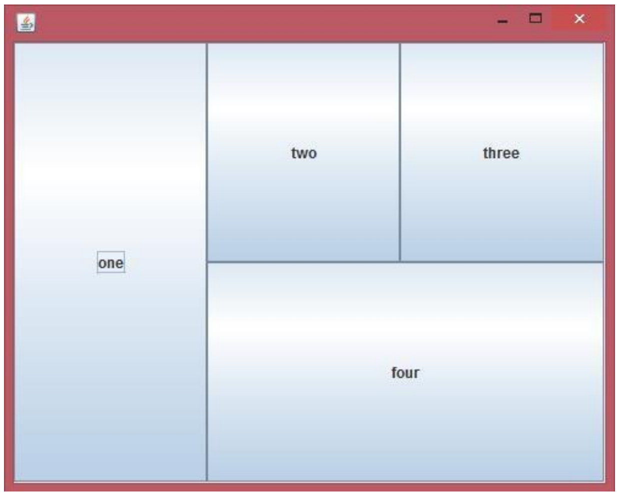
Spanning rows and columns

Perhaps the best feature of the `GridBaglayout` is that it lets you create arrangements in which components span two or more rows or columns. To do so, you set the `gridwidth` and `gridheight` variables of the `GridBagConstraints`.





```
setLayout( new GridBagLayout() );
Container c =getContentPane();
GridBagConstraints constraints = new GridBagConstraints();
constraints.weightx = 1.0;
constraints.weighty = 1.0;
constraints.fill = GridBagConstraints.BOTH;

constraints.gridwidth=1; // same as default
constraints.gridheight = 2; // Span two rows
JButton b1=new JButton("one");
constraints.gridx=0; constraints.gridy=0;
c.add(b1, constraints);

constraints.gridwidth=1;
constraints.gridheight = 1;
JButton b2=new JButton("two");
constraints.gridx=1; constraints.gridy=0;
c.add(b2, constraints);

constraints.gridwidth=1;
constraints.gridheight = 1;
JButton b3=new JButton("three");
constraints.gridx=2; constraints.gridy=0;
c.add(b3, constraints);



constraints.gridwidth = 2; // Span two columns
constraints.gridheight = 1;
JButton b4=new JButton("four");
constraints.gridx=1; constraints.gridy=1;
c.add(b4, constraints);
```



The output of the above program

The size of each element is controlled by the `gridwidth` and `gridheight` values of its constraints. For button one, we set `gridheight` to 2. Therefore, it is two cells high; its `gridx` and `gridy` positions are both zero, so it occupies cell (0,0) and the cell directly below it, (0,1). Likewise, button four has a `gridwidth` of 2 and a `gridheight` of 1, so it occupies two cells horizontally. We place this button in cell (1,1), so it occupies that cell and its neighbor, (2,1).

The null Layout Manager

To remove a layout manager, set the layout manager to null

```
myContainer.setLayout(null);
```

The following code sets the layout manager of a JFrame's content pane to null.

```
JFrame  frame  = new JFrame("");

Container contentPane = frame.getContentPane();

contentPane.setLayout(null);
```

"null layout manager" is also known as absolute positioning.

The following code shows how to use a null layout manager for the content pane of a JFrame. It layouts two buttons to it using the setBounds() method.

```
package example3;
import java.awt.Container;
import javax.swing.JButton;
import javax.swing.JFrame;

public class Example3 extends JFrame{
    public Example3() {


    Container c = getContentPane();
    c.setLayout(null);

    JButton b1 = new JButton("Button");
    JButton b2 = new JButton("2");
    c.add(b1);
    c.add(b2);

    b1.setBounds(100, 10, 200, 20);
    b2.setBounds(520, 10, 50, 40);

    setBounds(0, 0, 650, 500);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
    }


  public static void main(String[] args) {
    Example3 frame = new Example3();

  }
}
```

In this case, we need to align each component in an absolute position and avoiding the overlapping.

```
package lecture1ex1;

import javax.swing.JFrame;

public class Lecture1 extends JFrame {

      public Lecture1() {
        setTitle("My First Java Example with Netbeans");
        setSize(500,600);
        setLocation(400,700);
       setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        setVisible(true);
    }

    public static void main(String[] args) {
     new Lecture1();
    }

 }
}
```

```
package lecture1ex2;
import java.awt.Dimension;
import java.awt.Toolkit;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Container;
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;

public class lecture2 extends JFrame implements ActionListener {
        public Lecture2() {
        setTitle("My second Java Example with Netbeans");

        Dimension d=Toolkit.getDefaultToolkit().getScreenSize();
        setSize(d.width*4/5,d.height*4/5);

        Dimension d1=Toolkit.getDefaultToolkit().getScreenSize();
        Dimension d2=getSize();
        setLocation(500,800);
        setLayout( new FlowLayout() );
        JButton btn1 = new JButton("click me");
        contents.add(btn1);
       btn1.addActionListener(this);

        JButton btn2 = new JButton("exit");
        contents.add(btn2);

        btn2.addActionListener(this);

       setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        setVisible(true);
    }

    public static void main(String[] args) {
     new Lecture2();
    }
    }

}
```