

Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Before learning the Java abstract class, let's understand the abstraction in Java first.

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Example of abstract class

1. **abstract class** A{}
-

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

Example of abstract method

1. **abstract void** printStatus();//no method body and abstract
-

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

1. **abstract class** Bike{
2. **abstract void** run();
3. }
4. **class** Honda4 **extends** Bike{
5. **void** run(){System.out.println("running safely");}
6. **public static void** main(String args[]){
7. Bike obj = **new** Honda4();
8. obj.run();
9. }
10. }

```
running safely
```

Understanding the real scenario of Abstract class

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

Mostly, we don't know about the implementation class (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.

A **factory method** is a method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

File: TestAbstraction1.java


```

1. abstract class Shape{
2. abstract void draw();
3. }
4. //In real scenario, implementation is provided by others i.e. unknown by end user
5. class Rectangle extends Shape{
6. void draw(){System.out.println("drawing rectangle");}
7. }
8. class Circle1 extends Shape{
9. void draw(){System.out.println("drawing circle");}
10. }
11. //In real scenario, method is called by programmer or user
12. class TestAbstraction1{
13. public static void main(String args[]){
14. Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape()
    method
15. s.draw();
16. }
17. }

```

Another example of Abstract class in java

File: TestBank.java

```

1. abstract class Bank{
2. abstract int getRateOfInterest();
3. }
4. class SBI extends Bank{
5. int getRateOfInterest(){return 7;}
6. }
7. class PNB extends Bank{
8. int getRateOfInterest(){return 8;}
9. }
10.
11. class TestBank{
12. public static void main(String args[]){
13. Bank b;
14. b=new SBI();
15. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
16. b=new PNB();
17. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
18. }}

```

Test it Now

```

Rate of Interest is: 7 %
Rate of Interest is: 8 %

```

Abstract class having constructor, data member and methods

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

File: TestAbstraction2.java

```
1. //Example of an abstract class that has abstract and non-abstract methods
2. abstract class Bike{
3.     Bike(){System.out.println("bike is created");}
4.     abstract void run();
5.     void changeGear(){System.out.println("gear changed");}
6. }
7. //Creating a Child class which inherits Abstract class
8. class Honda extends Bike{
9.     void run(){System.out.println("running safely..");}
10. }
11. //Creating a Test class which calls abstract and non-abstract methods
12. class TestAbstraction2{
13.     public static void main(String args[]){
14.         Bike obj = new Honda();
15.         obj.run();
16.         obj.changeGear();
17.     }
18. }
```

Test it Now

```
bike is created
running safely..
gear changed
```

Rule: If there is an abstract method in a class, that class must be abstract.

```
1. class Bike12{
2.     abstract void run();
3. }
```

Test it Now

```
compile time error
```

Rule: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

Note: If you are beginner to java, learn interface first and skip this example.

```
1. interface A{
2.   void a();
3.   void b();
4.   void c();
5.   void d();
6. }
7.
8. abstract class B implements A{
9.   public void c(){System.out.println("I am c");}
10. }
11.
12. class M extends B{
13.   public void a(){System.out.println("I am a");}
14.   public void b(){System.out.println("I am b");}
15.   public void d(){System.out.println("I am d");}
16. }
17.
18. class Test5{
19.   public static void main(String args[]){
20.     A a=new M();
21.     a.a();
22.     a.b();
23.     a.c();
24.     a.d();
25.   }}
```

Test it Now

```
Output:I am a
       I am b
       I am c
       I am d
```

```
class CommandLineExample{  
public static void main(String args[]){  
System.out.println("Your first argument is: "+args[0]);  
}  
}
```

compile by > javac CommandLineExample.java
run by > java CommandLineExample sonoo

```
class A{  
public static void main(String args[]){  
  
for(int i=0;i<args.length;i++)  
System.out.println(args[i]);  
  
}  
}
```

compile by > javac A.java
run by > java A sonoo jaiswal 1 3 abc

```
Output: sonoo  
       jaiswal  
       1  
       3  
       abc
```

```
import java.lang.*;
import java.io.*;
import java.util.*;
class palindrome
{
    public static void main(String args[])
    {
        boolean flag=true;
        String str;
        str=args[0];
        int len= str.length();
        System.out.println("Length: "+len);
        for(int i=0;i<(len/2);i++)
        {
            if(str.charAt(i)==str.charAt((len-1)-i)) ;
            else { flag=false; break;}
        }
        if(flag==true) System.out.println("Palindrome");
        else System.out.println("Not Palindrome");
    }
}
```

An **interface in java** is a blueprint of a class. It has static constants and abstract methods.

The interface in java is a **mechanism to achieve abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve abstraction and multiple inheritance in Java.

Java Interface also **represents IS-A relationship**.

It cannot be instantiated just like abstract class.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

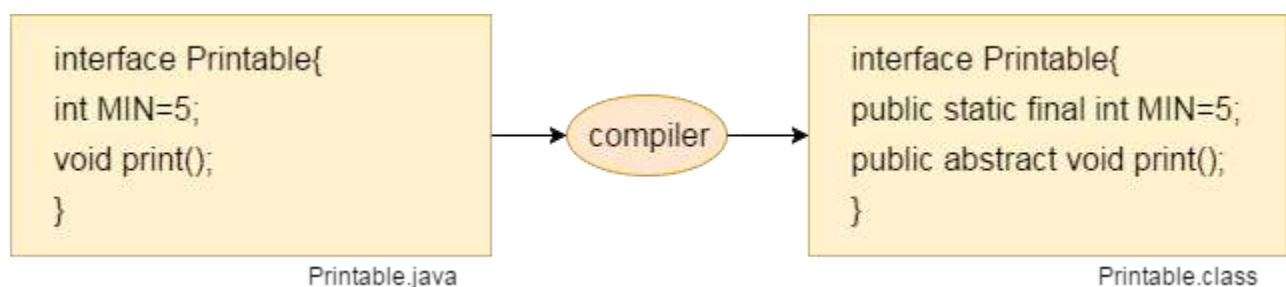
Java 8 Interface Improvement

Since Java 8, interface can have default and static methods which is discussed later.

Internal addition by compiler

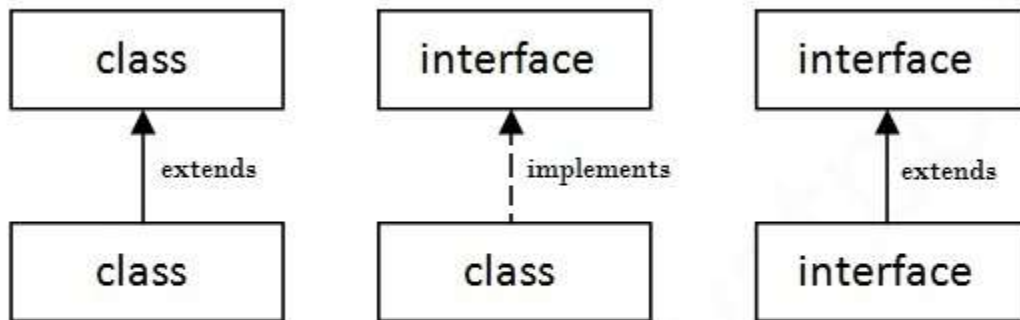
The java compiler adds **public** and **abstract** keywords before the interface method. More, it adds **public**, **static** and **final** keywords before data members.

In other words, Interface fields are public, static and final by default, and methods are public and abstract.



Understanding relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



Java Interface Example

In this example, Printable interface has only one method, its implementation is provided in the A class.

```
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
}
}
```

Output:

```
Hello
```

Java Interface Example: Drawable

In this example, Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In real scenario, interface is defined by someone but implementation is provided by different implementation providers. And, it is used by someone else. The implementation part is hidden by the user which uses the interface.

File: TestInterface1.java

//Interface declaration: by first user

```
interface Drawable{  
void draw();  
}
```

//Implementation: by second user

```
class Rectangle implements Drawable{  
public void draw(){System.out.println("drawing rectangle");}  
}  
class Circle implements Drawable{  
public void draw(){System.out.println("drawing circle");}  
}
```

//Using interface: by third user

```
class TestInterface1{  
public static void main(String args[]){  
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()  
d.draw();  
}}
```

Test it Now

Output:

```
drawing circle
```

Java Interface Example: Bank

Let's see another example of java interface which provides the implementation of Bank interface.

File: TestInterface2.java

```
interface Bank{  
float rateOfInterest();  
}  
class SBI implements Bank{
```



```

public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}}

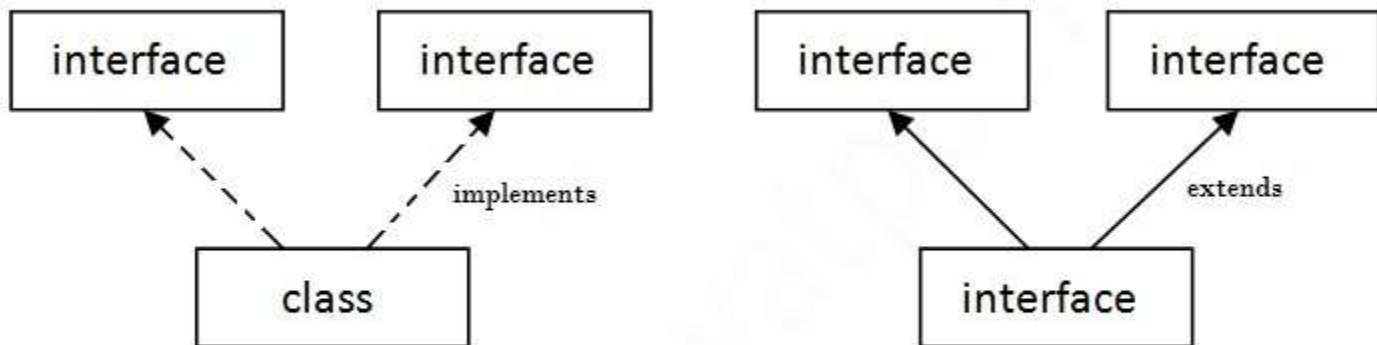
```

Output:

```
ROI: 9.15
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.



Multiple Inheritance in Java

```

interface Printable{
void print();
}
interface Showable{
void show();
}

```

```

}
class A7 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
}
}

```

```

Output:Hello
       Welcome

```

Q) Multiple inheritance is not supported through class in java but it is possible by interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class because of ambiguity. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

```

interface Printable{
void print();
}
interface Showable{
void print();
}

class TestInterface3 implements Printable, Showable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
TestInterface3 obj = new TestInterface3();
obj.print();
}
}

```

Output:

```
Hello
```

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestInterface1, so there is no ambiguity.

Interface inheritance

A class implements interface but one interface extends another interface .

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();
}
}
```

Output:

```
Hello
Welcome
```

Java 8 Default Method in Interface

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

File: TestInterfaceDefault.java

```

interface Drawable{
void draw();
default void msg(){System.out.println("default method");}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class TestInterfaceDefault{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
d.msg();
}}

```

Output:

```

drawing rectangle
default method

```

Java 8 Static Method in Interface

Since Java 8, we can have static method in interface. Let's see an example:

File: TestInterfaceStatic.java

```

interface Drawable{
void draw();
static int cube(int x){return x*x*x;}
}
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}

class TestInterfaceStatic{
public static void main(String args[]){
Drawable d=new Rectangle();
d.draw();
System.out.println(Drawable.cube(3));
}
}

```

```
}}
```

Output:

```
drawing rectangle  
27
```

Q) What is marker or tagged interface?

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

//How Serializable interface is written?

```
public interface Serializable{  
}
```

Nested Interface in Java

Note: An interface can have another interface i.e. known as nested interface. We will learn it in detail in the nested classes chapter. For example:

```
interface printable{  
    void print();  
    interface MessagePrintable{  
        void msg();  
    }  
}
```

[More about Nested Interface](#)

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. S have default and static methods also.
2) Abstract class doesn't support multiple inheritance .	Interface supports multiple inheritance .
3) Abstract class can have final, non-final, static and non-static variables .	Interface has only static and final variables .
4) Abstract class can provide the implementation of interface .	Interface can't provide the implementation class .
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
6) Example: <pre>public abstract class Shape{ public abstract void draw(); }</pre>	Example: <pre>public interface Shape{ void draw(); }</pre>

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

1. *//Creating interface that has 4 methods*
2. **interface** A{
3. **void** a();//bydefault, public and abstract
4. **void** b();
5. **void** c();
6. **void** d();

```

7. }
8.
9. //Creating abstract class that provides the implementation of one method of A interface
10. abstract class B implements A{
11. public void c(){System.out.println("I am C");}
12. }
13.
14. //Creating subclass of abstract class, now we need to provide the implementation of rest of
    the methods
15. class M extends B{
16. public void a(){System.out.println("I am a");}
17. public void b(){System.out.println("I am b");}
18. public void d(){System.out.println("I am d");}
19. }
20.
21. //Creating a test class that calls the methods of A interface
22. class Test5{
23. public static void main(String args[]){
24. A a=new M();
25. a.a();
26. a.b();
27. a.c();
28. a.d();
29. }}

```

Test it Now

Output:

```

I am a
I am b
I am c
I am d

```

A **java package** is a group of similar types of classes, interfaces and sub-packages.

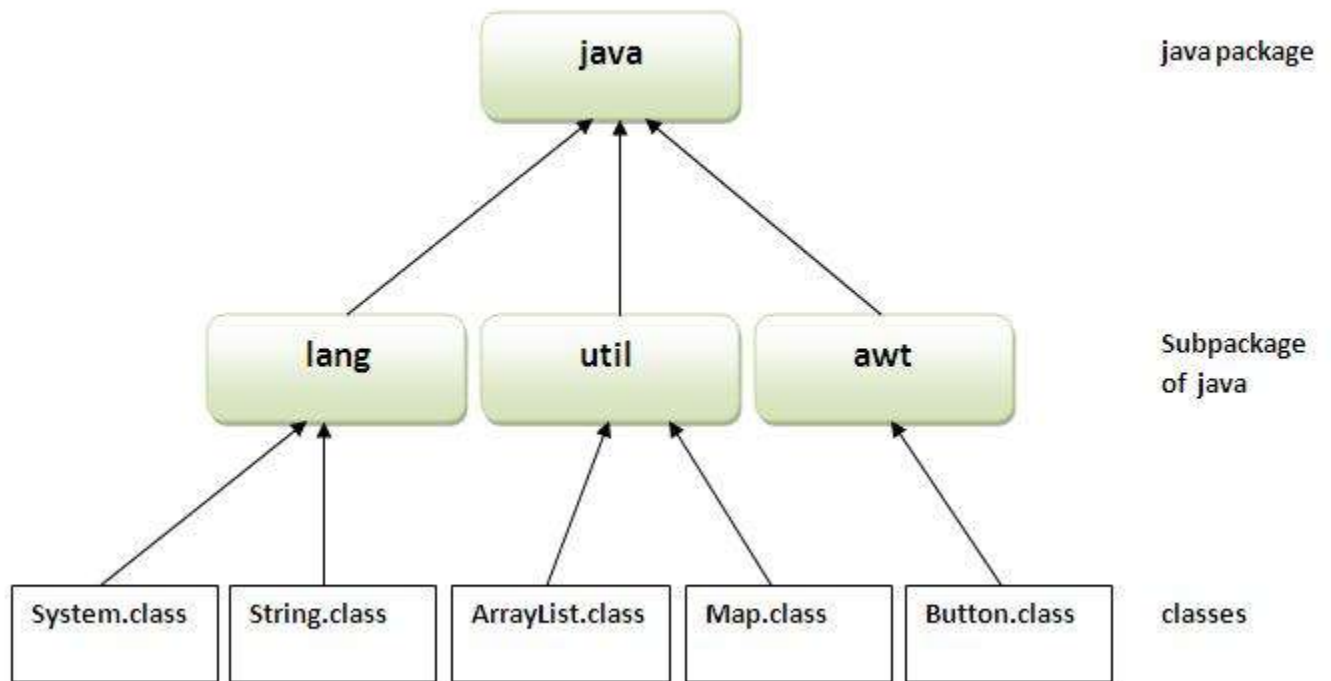
Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

- 1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- 2) Java package provides access protection.
- 3) Java package removes naming collision.



Simple example of java package

The **package keyword** is used to create a package in java.

1. `//save as Simple.java`
2. `package mypack;`
3. `public class Simple{`
4. `public static void main(String args[]){`
5. `System.out.println("Welcome to package");`
6. `}`
7. `}`

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

1. `javac -d directory javafilename`

For **example**

1. `javac -d . Simple.java`

The `-d` switch specifies the destination where to put the generated class file. You can use any directory name like `/home` (in case of Linux), `d:/abc` (in case of windows) etc. If you want to keep the package within the same directory, you can use `.` (dot).

How to run java package program

You need to use fully qualified name e.g. `mypack.Simple` etc to run the class.

To Compile: `javac -d . Simple.java`

To Run: `java mypack.Simple`

Output:Welcome to package

The `-d` is a switch that tells the compiler where to put the class file i.e. it represents destination. The `.` represents the current folder.

How to access package from another package?

There are three ways to access the package from outside the package.

1. `import package.*;`
2. `import package.classname;`
3. `fully qualified name.`

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

//save by A.java

```
package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
```

//save by B.java

```
package mypack;
import pack.*;
```

```
class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

Output:Hello

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

//save by A.java

```
package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
```

//save by B.java

```
package mypack;
```

```
import pack.A;
```

```
class B{  
    public static void main(String args[]){  
        A obj = new A();  
        obj.msg();  
    }  
}
```

```
Output:Hello
```

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java
```

```
package pack;  
public class A{  
    public void msg(){System.out.println("Hello");}  
}
```

```
//save by B.java
```

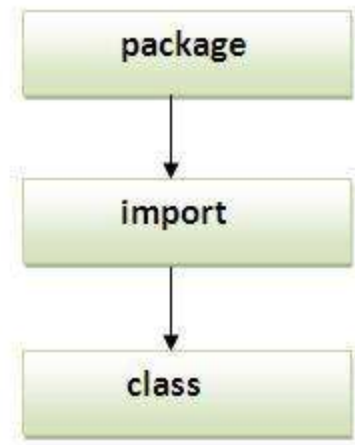
```
package mypack;  
class B{  
    public static void main(String args[]){  
        pack.A obj = new pack.A();//using fully qualified name  
        obj.msg();  
    }  
}
```

```
Output:Hello
```

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.



Applet vs Application

Applet

- Applets are small java programs that are designed to be included in a HTML web document. They require a java enabled browser for execution
- Applets do not use main() method
- Cannot communicate with other servers
- Applets have no disk & network access
- Cannot run independently it requires API's (Ex: web API)

Application

- Applications are stand alone programs that can be run independently without having to use web browsers.
- uses the main() method for execution
- Communication with other servers is probably possible
- Java Applications have access to local file system & network
- can run alone but require JRE

⇒ Displaying Graphics in Applet

Java AWT Graphics class provides many methods for graphics programming.

Commonly used methods of Graphics class

- (i) public abstract void drawString (String s, int x, int y) :-
it is used to draw specified string

- (ii) `public void drawRect(int x, int y, int width, int height);`
is used to draw Rectangle with specified width & height.
- (iii) `public abstract void fillRect(int x, int y, int width, int height);`
used to fill rectangle with default color & specified width & height.
- (iv) `public abstract void draw oval(int x, int y, int width, int height);`
- (v) `public abstract void fill oval(int x, int y, int width, int height);`
- (vi) `public abstract void drawLine(int x, int y, int x2, int y2);`
is used to draw line between the points (x_1, y_1) & (x_2, y_2)
- (vii) `public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle);` ÷ used to draw circular or elliptical arc
- (viii) `public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle);`
- (ix) `public abstract void setColor(Color c);` ÷ used to set the graphics current color to specified color.

/*Program demonstrating methods in Graphics class */

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

public class GraphicsDemo1 extends Applet
{
    public void paint(Graphics g)
    {
        g.setColor(Color.blue);
    }
}
```

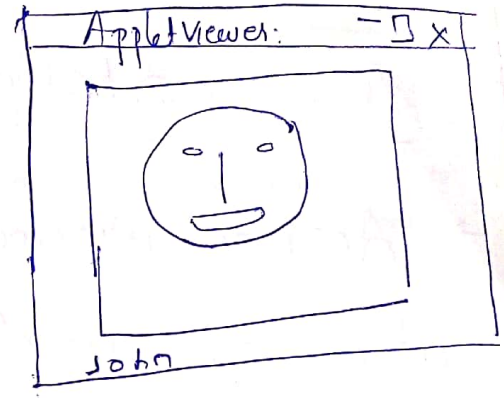


```

g. fillRect (40, 40, 200, 200);
g. setColor (Color. yellow);
// face
g. fillOval (90, 70, 80, 80);
g. setColor (Color. black);
// set eyes
g. fillOval (110, 95, 5, 5);
g. fillOval (145, 95, 5, 5);
g. drawLine (130, 95, 130, 115);
g. setColor (Color. red);
// mouth
g. fillArc (113, 115, 35, 20, 0, -180);
g. drawString ("john", 40, 340);
}

```

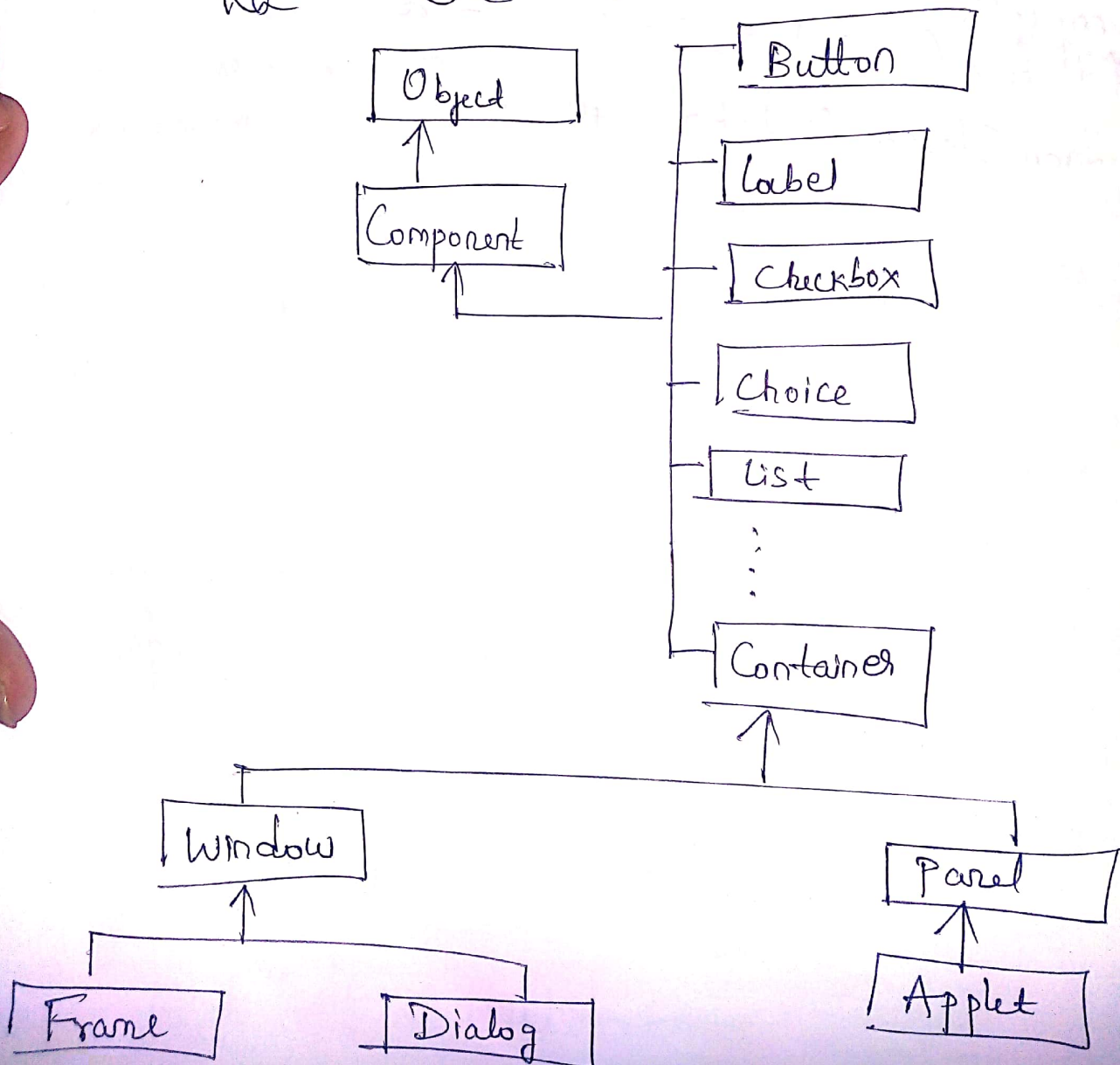
o/p



Introduction to Graphics programming using AWT (Abstract Window Toolkit)

→ Abstract Window Toolkit (AWT) represents a class library to develop applications using GUI. The java.awt package got classes & interfaces to develop GUI (Graphical user interface) & let users interact with applications.

fig: Classes of AWT (AWT class hierarchy)



using
a class
The
develop
applications.
y)

Component ÷ All the elements like buttons, textfields, scrollbars etc. are known as Components. Component represents an object which is displayed pictorially on the screen. For example, we create an object of Button class:

```
Button b = new Button ( );
```

Now, b is object of Button class. If we display this b on the screen, it displays push button.

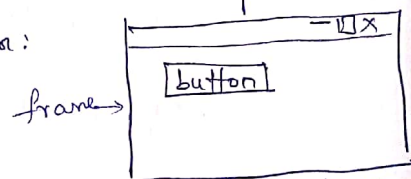
Container ÷ A Container is like a screen where we add Components like button, textfields etc. It is a Component in AWT that contain another Components like button etc. There are 4 types of Containers in AWT:

→ Window, Frame, Dialog & Panel.

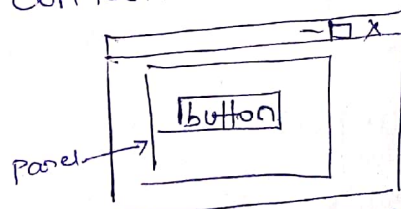
Window ÷ Window is a container that represents imaginary rectangular area on screen without any borders or menus.

Frame ÷ Frame is a container that contain title bar, border & menu bars. It can contain Components like button, textfield etc.

Ex:



Panel: Panel is a container which offers space to place any other Component. panel doesn't contain titlebar, menu or border.



Dialog: Dialog class has border & title. An instance of Dialog class cannot exist without an associated instance of Frame class.

→ Creating a Frame :

A Frame becomes the basic Component of AWT. The frame has to be created before any other Component as all other Components can be displayed ~~before~~ in a frame.

There are three ways to create a Frame.

① Create a Frame class object

Frame f = new Frame();

② Create a Frame class object & pass its title

Frame f = new Frame("my frame");

③ Create a Subclass MyFrame to the Frame class & Create an object to the Subclass as (i.e. By extending Frame class)

class MyFrame extends Frame

MyFrame f = new MyFrame();

Program to create a frame by creating an object to frame class

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

```
class MyFrame
```

```
{
```

```
    public static void main (String args[])
```

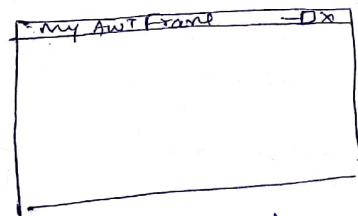
```
}
```

of AWT.
any other component
before

create a Frame:

```
// Create a Frame
Frame f = new Frame("My AWT Frame");
// Set size of the frame
f.setSize(300, 250);
f.setVisible(true); // display the frame
```

o/p:



```
o/p:
c:\> javac MyFrame.java
c:\> java MyFrame
```

/* program to create a Frame by creating object to the
subclass of Frame */

```
import java.awt.*;
class MyFrame extends Frame
```

```
{
    // call superclass constructor to store title
```

```
MyFrame(String str)
```

```
{
    super(str);
```

```
}
public static void main(String args[])
```

```
{
    // Create a frame with title
```

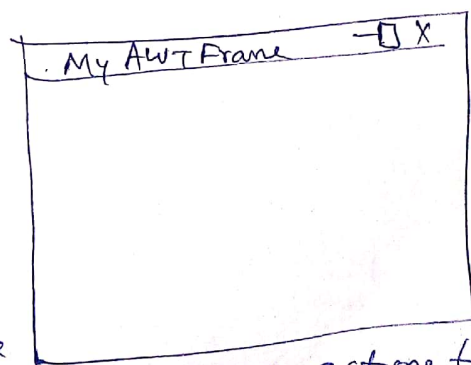
```
MyFrame f = new MyFrame("My AWT Frame");
```

```
f.setSize(300, 250);
```

```
f.setVisible(true);
```

```
} }
```

o/p:



note: This frame can be minimized, maximized & resized, but cannot be closed. Closing frame is possible by attaching action to component. To attach actions to the components, we need 'event delegation model'.

Note:

Some important methods of Component class

- void setSize (int width, int height) : set size of frame.
- void setVisible (boolean b) ÷ makes GUI Component visible to user depending on Boolean parameter pass in function by default false.
- void add (Component c) ÷ Add a Component to Container.
- void setLayout (LayoutManager m) : defines the layout manager for Component.

// program demonstrating to add Component in Container.

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

```
class addcomponent extends Frame
```

```
{
```

```
    addComponent()
```

```
{
```

```
    Button b = new Button("Click on me");
```

```
    b.setBounds(30, 90, 90, 30);
```

```
    setSize(200, 300);
```

```
    setVisible(true);
```

```
    setLayout(null);
```

```
    add(b);
```

```
}
```

```
public static void main(String args[])
```

```
{
```

```
    addcomponent a = new addcomponent();
```

```
}
```

```
}
```

Event-driven Programming:

Event: Change in state of object is known as event.

Any program that GUI written for windows is event driven.

Event Delegation Model:

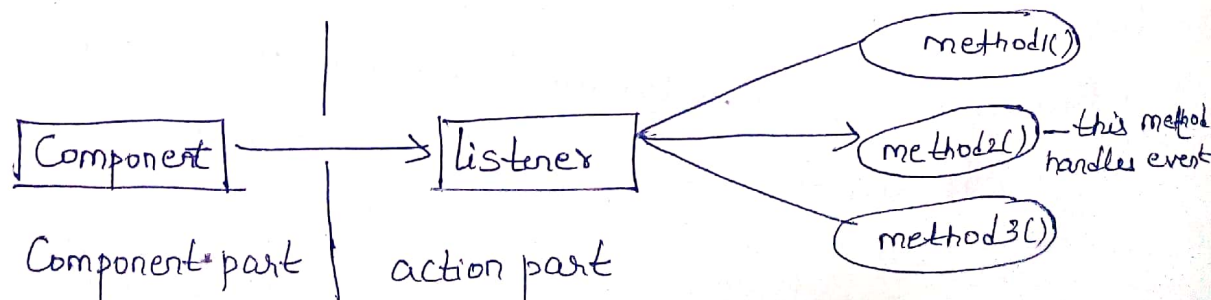
When we create a component, generally the component is displayed on the screen but it is not capable of performing any action. For example, we created a push button, which can be displayed but cannot perform any action, even when someone click on it. Clicking like this is ^{called} an event.

- An event represents a specific action done on a component. Clicking, typing ~~in~~ data inside the component, moving mouse over etc are all examples of event.

→ When an event is generated on the component, the component will not know about it, because it cannot listen to the event. So we should add some listener to the components.

A listener is an interface which listens to an event coming from a component. A listener will have some abstract methods which need to be implemented by programmer.

- Event delegation model represents that when an event is generated by the user on a component, it is delegated to a listener interface & the listener calls a method in response to the event. Finally, the event is handled by the method



Components of Event handling:-

- (i) Event :- An event is a change in state of object
- (ii) Source :- Event source is an object that generates an event
- (iii) Listener :- it is also known as eventhandler. Listener is responsible for generating response to an event.

Steps involved in event delegation model:

- (i) We should attach an appropriate listener to a component. This is done using `addxxxListener()` method. Similarly, to remove a listener from a component, we can use `removexxxListener()` method.
- (ii) Implement the methods of the listener, especially the method which handles the event.
- (iii) When an event is generated on the component, then the method in step 2 will be executed & the event is handled.

Ex: Closing the Frame:

— Steps to use event delegation model to close the Frame:

- (i) attach a listener to the frame component. All listeners are available in `java.awt` package. The most suitable listener to the frame is `WindowListener`. It can be attached using `addWindowListener()` method as
f. `addWindowListener (WindowListener obj);`

object
generates
er. Listener
event.

del:

s to

er ()

m a

od.

pecially

nt,

event

Frame:

terers

e

e

Note that addWindowListener () method has a parameter that is expecting object of WindowListener interface.

Since it is not possible to create an object to an interface, we should create an object to the implementation class of the interface & pass it to the method.

⇒ (i) Implement all the methods of the WindowListener interface. The following methods are found in WindowListener interface:

```
public void windowActivated (WindowEvent e)
public void windowClosed (WindowEvent e)
public void windowClosing (WindowEvent e)
public void windowDeactivated (WindowEvent e)
public void windowDeiconified (WindowEvent e)
public void windowIconified (WindowEvent e)
public void windowOpened (WindowEvent e)
```

→ In all the preceding methods, WindowListener interface calls the public void windowClosing () method when the frame is being closed. So implementing this method alone is enough.

```
public void windowClosing (WindowEvent e)
```

```
{
    // close the application
```

```
    System.exit(0);
```

```
}
```

For the remaining methods, we can provide empty body.

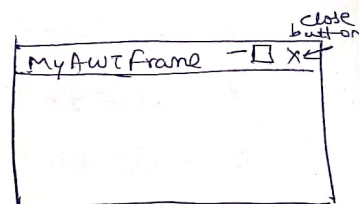
→ So, when the frame is closed, the body of this method is executed & the application gets closed. In this way, we can handle frame closing event.

/* Write a program which first creates a frame
& then closes it on clicking the close button */

```
import java.awt.*;  
import java.awt.event.*;  
class MyFrame extends Frame  
{  
    public static void main(String args[])  
    {  
        // create a frame with title  
        MyFrame f = new MyFrame();  
        // set a title for the frame  
        f.setTitle("My AWT Frame");  
        // set the size of the frame  
        f.setSize(300, 250);  
        // display the frame  
        f.setVisible(true);  
        // close the frame  
        f.addWindowListener(new Myclass());  
    }  
}  
class Myclass implements WindowListener  
{  
    public void windowActivated(WindowEvent e) {}  
    public void windowClosed(WindowEvent e) {}  
    public void windowClosing(WindowEvent e)  
    {  
        System.exit(0);  
    }  
}
```


frame

```
public void windowDeactivated(WindowEvent e) { }  
public void windowDeiconified(WindowEvent e) { }  
public void windowIconified(WindowEvent e) { }  
public void windowOpened(WindowEvent e) { }  
}  
o/p: C:\> javac MyFrame.java  
      java MyFrame
```



In this program, we not only create a frame but also close the frame when user clicks on the close button. For this purpose we use WindowListener interface. Here, we had mention all the methods of WindowListener interface, just for the sake of one method. Instead we can use a class WindowAdapter in java.awt.event package, that contains all the methods of the WindowListener interface with an empty implementation (body). WindowAdapter is an adapter class of WindowListener interface.

What is an adapter class?

- An adapter class is an implementation class of a listener interface which contains all methods implemented with empty body. It reduce overhead of programming while working with listener interfaces.

/ * program to close frame using WindowAdapter class */

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

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

```
class myframe extends Frame
```

```
{
```

```
Public static void main (String args [])
```

```
{
```

```
    MyFrame f = new MyFrame ();
```

```
    f . setTitle (" My AWT Frame ");
```

```
    f . setSize (300, 250);
```

```
    f . setVisible (true);
```

```
    f . addWindowListener (new Myclass());
```

```
} }
```

```
Class myclass extends WindowAdapter
```

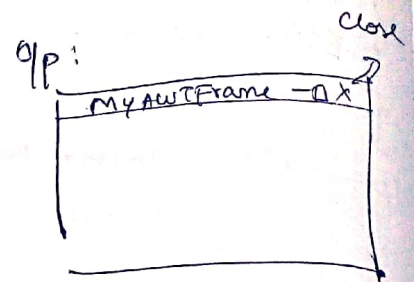
```
{
```

```
    public void windowClosing (WindowEvent e)
```

```
    {
```

```
        System . exit (0);
```

```
    } }
```



Note :- The code of myclass can be copied directly into addWindowListener () method as:

```
f . addWindowListener (new WindowAdapter ( )
```

```
{
```

```
    public void windowClosing (WindowEvent e)
```

```
    {
```

```
        System . exit (0);
```

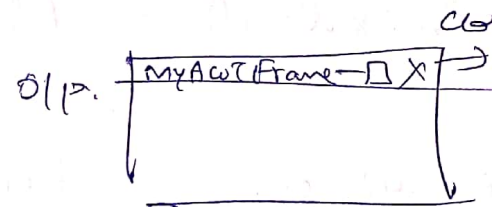
```
    } }
```

Anonymous inner class is an inner class whose name is not mentioned, & for which only one object is created.

```

/* program to close the frame using an
anonymous class */
import java.awt.*;
import java.awt.event.*;
class MyFrame extends Frame
{
    public static void main(String args[])
    {
        MyFrame f = new MyFrame();
        f.setTitle("My AWT Frame");
        f.setSize(300, 250);
        f.setVisible(true);
        f.addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
    }
}

```



Important Event classes & their interface

<u>Event class</u>	<u>Sources of Event</u>	<u>Description :</u>	
→ ActionEvent	Button, List, Menu	Generated when a button is pressed, a list-item is double clicked, or a menu item is selected.	→ 1
→ MouseEvent	Mouse	Generated when the mouse is dragged, moved, clicked, pressed or dragge released also generated when the mouse is enters or exits a component	→ 1 → 1
→ MouseWheelEvent	Mouse	Generated when mouse wheel is moved	→ 1
→ KeyEvent	Keyboard	Generated when input is received from keyboard	→ 1
→ TextEvent	TextField, Text Area	Generated when the value of a Text Area or TextField is changed	→ 1
→ AdjustmentEvent	Scrollbar	Generated when scrollbar is manipulated	→ 1
→ ComponentEvent	All components	Generated when a component is hidden, moved, resized or becomes visible	→ 1
→ ContainerEvent	All containers	Generated when a component is added or removed from a container.	→ 1

Interface

ion :
 a button is
 - item is
 or a menu
 cted.
 the mouse
 ed, clicked,
 released
 when the
 or exits

mouse wheel

it is recieved

value of
 is changed
 all bar is

component
 sized or

Component
 d from

Event class	Sources of Event	Description
→ WindowEvent	Window	Generated when window is activated, closed, deactivated, deiconified, iconified, opened, quit
→ FocusEvent	All components	Generated when a component gains or loses keyboard focus
→ InputEvent	Input Devices	Abstract super class for all component input classes
→ ItemEvent	Checkbox, Menu, List Choice	Generated when checkbox or list item is clicked, also occurs when a choice selection is made or a checkable menu item is selected or deselected.

Interfaces of event class & Methods.

Event	Listener of the event class	Methods of in the listener
• ActionEvent	Action Listener	public void actionPerformed(ActionEvent ae)
• MouseEvent	MouseListener, MouseWheelListener	public void mouseClicked(MouseEvent me) public void mouseEntered(MouseEvent me) public void mouseExited(MouseEvent me) public void mousePressed(MouseEvent me) public void mouseReleased(MouseEvent me)
• MouseEvent	Mouse Motion Listener	public void mouseDragged(MouseEvent me) public void mouseMoved(MouseEvent me) public void mouseWheelMoved(MouseWheelEvent me)

<u>Event class</u>	<u>Listener of event class</u>	<u>Methods in the Listener</u>
KeyEvent	KeyListener	public void keyPressed(KeyEvent ke) public void keyReleased(KeyEvent ke) public void keyTyped(KeyEvent ke)
TextEvent	TextListener	public void textChanged(TextEvent te)
AdjustmentEvent	AdjustmentListener	public void adjustmentValueChanged(AdjustmentEvent ae)
ComponentEvent	ComponentListener	public void componentResized(ComponentEvent ce) " " componentMoved(" " " " " componentShown(" " " " " componentHidden(" " ")
ContainerEvent	ContainerListener	public void componentAdded(ContainerEvent ce) public void componentRemoved(ContainerEvent ce)
WindowEvent	WindowListener	public void windowActivated(WindowEvent ae) " " windowClosed(" " " " " windowClosing(" " " " " windowDeactivated(" " " " " windowIconified(" " " " " windowDeiconified(" " " " " windowOpened(" " ")
FocusEvent	FocusListener	public void focusGained(FocusEvent fe) public void focusLost(FocusEvent fe)
ItemEvent	ItemListener	public void itemStateChanged((ItemEvent ie)

/* Develop a program for the demonstration of

Creating a button */

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

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

```
public class ButtonExample
```

```
{  
    public static void main (String args [])
```

```
{  
    Frame f = new Frame("Button Example");
```

```
    final TextField tf = new TextField();
```

```
    tf.setBounds(50, 50, 50, 20);
```

```
    Button b = new Button("Click here");
```

```
    b.setBounds(50, 100, 60, 30);
```

```
    b.addActionListener(new ActionListener() {
```

```
    {  
        public void actionPerformed (ActionEvent e)
```

```
    {  
        tf.setText("Welcome to java awt");
```

```
    }  
    }  
});
```

```
f.add(b);
```

```
f.add(tf);
```

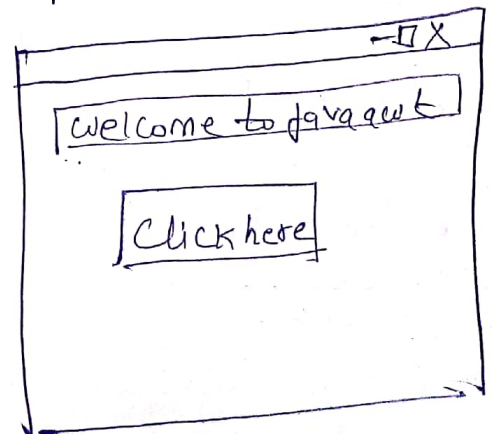
```
f.setSize(400, 400);
```

```
f.setLayout(null);
```

```
f.setVisible(true);
```

```
}
```

O/P:



Button:

A button is a component which contains a label & generates an event when pressed. Buttons are objects of class Button. Button defines these two

Constructors:

- Button() : creates an empty button.
- Button(String str) : creates a button that contains str as a label.

TextField:

The TextField class implements a single line entry area, usually called as edit control. TextField is subclass of TextComponent

TextField has following Constructors:

- TextField() : creates default text field
- TextField(int no. of chars) : This will create a text field & will accept the no. of characters specified
- TextField(String str) : Creates TextField with String str
- TextField(String str, int) : initializes text & set its width.

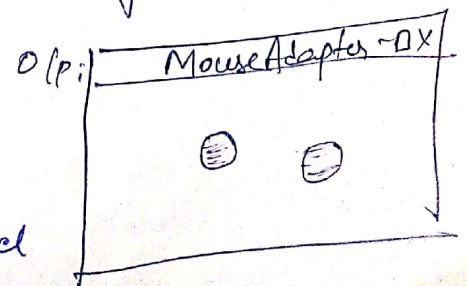
Methods:

- String getText(): gets the text from textfield
- void setText(String str): to set the Text

/* Program to demonstrate mouse event handling

```
import java.awt.*;  
import java.awt.event.*;  
public class MA extends MouseAdapter  
{  
    Frame f = new Frame();  
    MA()  
    {  
        f = new Frame("MouseAdapter");  
        f.addMouseListeners(this);  
        f.setSize(300, 300);  
        f.setLayout(null);  
        f.setVisible(true);  
    }  
    public void mouseClicked(MouseEvent e)  
    {  
        Graphics g = f.getGraphics();  
        g.setColor(Color.BLUE);  
        g.fillOval(e.getX(), e.getY(), 30, 30);  
    }  
    public static void main(String args[])  
    {  
        MA m = new MA();  
    }  
}
```

Note: Shapes are drawn on Drawing panel
using an object named Graphics
Graphics g = panel.getGraphics();



Label :

The object of Label class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it display.

Label defines following constructors:

- Label() ÷ It creates a blank label
- Label(String str) ÷ It creates a label that contains the specified ^{str} by str.
- Label(String str, int pos) ÷ It creates a label that contains the string specified by str using the alignment specified by pos.

The value of pos must be one of these three

Constants:

Label.CENTER or
Label.RIGHT or
Label.LEFT

→ Graphics programming - Layout Managers :

- The Layout Managers are used to arrange components in a particular manner. It is an interface that is implemented by all classes of Layout Managers.

The following classes represent the layout managers:

- (i) FlowLayout
- (ii) BorderLayout
- (iii) CardLayout
- (iv) GridLayout
- (v) GridBagLayout
- (vi) BoxLayout

FlowLayout

— The FlowLayout is used to arrange the components in a line, one after another. It is default layout in applets & panel.

To create FlowLayout, we can use following ways:

(i) FlowLayout obj = new FlowLayout();

- This creates Flow layout. By default, the gap between components will be 5 pixels & the components are centered in first line.

(ii) FlowLayout obj = new FlowLayout(int alignment);

Here, the alignment of components can be specified.

→ To arrange the components starting from left to right we use FlowLayout.LEFT

→ To adjust the components towards right, we use

FlowLayout.RIGHT & for

→ center alignment, we use FlowLayout.CENTER

(iii) FlowLayout obj = new FlowLayout (int alignment,
int hgap, int vgap);

Here hgap & vgap specify the space between components
hgap represents horizontal gap & vgap represents
vertical gap in pixels.

/ * program to create a group of buttons &
arrange them in the ~~container~~ container
using FlowLayout manager. */

```
import java.awt.*; *;  
import java.awt.event.*; *;  
public class MyFlowLayout  
{  
    MyFlowLayout()
```

```
{  
    Frame f = new Frame();  
    Button b1 = new Button("1");  
    Button b2 = new Button("2");  
    Button b3 = new Button("3");  
    Button b4 = new Button("4");  
    Button b5 = new Button("5");
```

```
    f.add(b1);  
    f.add(b2);  
    f.add(b3);
```

f.add
f.ad
f.se

f.se
f.se
}
public
{
 M
}
}
Notes: To se
to layout c

Border
→ Border
the 4
border
→ The
as 'E
The c
Const
(i) Bor
(ii) Thi
bet
(iii) Bor
Cre
Ve

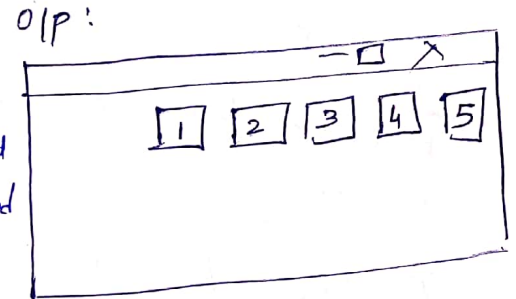

```
f.add(b4);
f.add(b5);
f.setLayout(new FlowLayout(FlowLayout.RIGHT,
                             10, 10));
```

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

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

```
}
}
```

Notes: To set particular layout, we should create object to layout class & pass the object to setLayout() method
 ex: FlowLayout obj = new FlowLayout();
 f.setLayout(obj); // f is contained.



Border Layout :

→ BorderLayout is useful to arrange the components in the 4 borders of the frame as well as in center. The borders are identified with the names of directions.
 → The top border is specified as 'North', the right border as 'East', the bottom one as 'South' & left one as 'West'.

The center is represented as 'Center'.

Constructors of BorderLayout class ?

- ① BorderLayout obj = new BorderLayout();
 This creates a BorderLayout object without any gaps between the components.
- ② BorderLayout obj = new BorderLayout(int hgap, int vgap);
 Creates a BorderLayout with given horizontal & vertical gaps between the component.

/* program to create a group of push button
+ add them to the container by using BorderLayout

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

```
public class Border
```

```
{
```

```
Frame Border()
```

```
{
```

```
Frame f = new Frame();
```

```
Button b1 = new Button("NORTH");
```

```
Button b2 = new Button("SOUTH");
```

```
Button b3 = new Button("EAST");
```

```
Button b4 = new Button("WEST");
```

```
Button b5 = new Button("CENTER");
```

```
f.add(b1, BorderLayout.NORTH);
```

```
f.add(b2, BorderLayout.SOUTH);
```

```
f.add(b3, BorderLayout.EAST);
```

```
f.add(b4, BorderLayout.WEST);
```

```
f.add(b5, BorderLayout.CENTER);
```

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

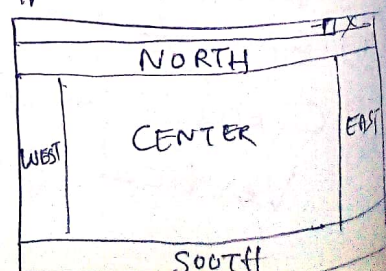
```
f.setVisible(true);
```

```
}  
public static void main (String[] args)
```

```
{  
Border b = new Border();
```

```
} }
```

o/p:



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) ÷ Create a grid layout with the given rows & columns but no gaps between the components

③ GridLayout(int rows, int columns, int hgap, int vgap) ÷ Creates a grid layout with given rows & columns along with given horizontal & vertical gaps.

Ex: /* Program to create button & add them to container by using BorderLayout */

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

```
public class BorderMyGrid
```

```
{ MyGrid  
  Border()
```

```
{ Frame f = new Frame();
```

```
  Button b1 = new Button("1");
```

```
  Button b2 = new Button("2");
```

```
  Button b3 = new Button("3");
```

```
  Button b4 = new Button("4");
```

```

Button b5 = new Button ("5");
Button b6 = new Button ("6");
Button b7 = new Button ("7");
Button b8 = new Button ("8");
Button b9 = new Button ("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));
f.setSize(300,300);
f.setVisible(true);

```

```

}
public static void main(String args[])
{
    MyGrid n = new MyGrid();
}
}

```

o/p:

- □ X		
1	2	3
4	5	6
7	8	9

→ Card Layout

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 & vertical gap.
- ② CardLayout(int hgap, int vgap): Creates a card layout with the given horizontal & vertical gap.

Commonly used methods of CardLayout class

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

/ * program to create a group of push button & add them to the container using CardLayout */

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

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

```
class CE extends Frame implements ActionListener
```

```
{
```

```
    CardLayout card = new CardLayout(20, 20);
```

CEC)

{

setLayout(card);

Button first = new Button("first");

Button second = new Button("second");

Button third = new Button("third");

add(first, "card1");

add(second, "card2");

add(third, "card3");

first.addActionListener(this);

second.addActionListener(this);

third.addActionListener(this);

}

public void actionPerformed(ActionEvent e)

{

card.next(this);

}

class CardLayoutExample

{

public static void main(String args[])

{

CE frame = new CEC;

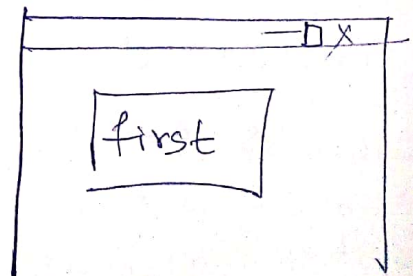
frame.setTitle("Cardlayout in java Example");

frame.setSize(220, 150);

frame.setResizable(false);

frame.setVisible(true);

}



/* Write a program which handles KeyboardEvent */

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

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

```
public class keypress extends Frame
```

```
{
```

```
    Label label;
```

```
    TextField txtfield;
```

```
    public static void main (String[] args)
```

```
{  
    KeyPress k = new keypress();
```

```
}
```

```
    public keypress()
```

```
{  
    Super("key press Event Frame");
```

```
    Panel panel = new Panel();
```

```
    panel.setBounds(40, 80, 200, 200);
```

```
    panel.setBackground(Color.gray);
```

```
    label = new Label();
```

```
    txtfield = new TextField(20);
```

```
    txtfield.addKeyListener(new MyKeyListener());
```

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

```
    panel.add(txtfield, BorderLayout.CENTER);
```

```
    add(panel, BorderLayout.CENTER);
```

```
    addWindowListener(new WindowAdapter())
```

```
{
```



```
public void windowClosing(WindowEvent we)
```

```
{
```

```
    System.exit(0);
```

```
}
```

```
});
```

```
setSize(400, 400);
```

```
setVisible(true);
```

```
}
```

```
public class MyKeyListener extends KeyAdapter
```

```
{  
    public void keyPressed(KeyEvent ke)
```

```
{  
    char i = ke.getKeyChar();
```

```
    String str = Character.toString(i);
```

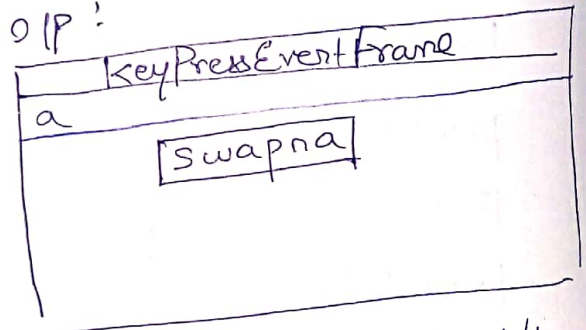
```
    label.setText(str);
```

```
}
```

```
}
```

```
}
```

O/P :



Note :

- Char getKeyChar(): Returns the character associated with the key in this event.

- Character toString(Char c) Method :-

The toString(Char c) method of Character class returns the String object which ^{represents} the given character's value.

/ * Program which handles Mouse Event with Mouse
with MouseMotionAdapter * /

```
import java.awt.* *;  
import java.awt.event.* *;  
class ME extends MouseMotionAdapter  
{  
    Frame f = new Frame("Mouse motion adapter");  
    MEC )  
    {  
        f.addMouseMotionListener(-this);  
        f.setSize(200, 300);  
        f.setVisible(true);  
        f.setLayout(null);  
    }  
    public void mouseDragged(MouseEvent e)  
    {  
        Graphics g = f.getGraphics();  
        g.setColor(Color.pink);  
        g.fillOval(e.getX(), e.getY(), 20, 20);  
    }  
    public static void main(String args[])  
    {  
        ME m = new ME();  
    }  
}
```

Check Boxes :

- A checkbox is a square shaped box which displays an option to the user. The user can select one or more options from a group of check boxes.

→ To create a checkbox, we create an object to checkbox class as:

→ `Checkbox cb = new Checkbox();` // creates checkbox without any label

→ `Checkbox cb = new Checkbox("label");` with a label

→ `Checkbox cb = new Checkbox("label", state);` // if state is true, then the checkbox appears as if it is selected by default, else not selected.

→ To get the state of checkbox:

`boolean b = cb.getState();`

If the checkbox is selected, this method returns true, else false.

→ To set the state of a checkbox

`cb.setState(true);`

The checkbox cb will now appear as if it is selected.

/ * program to create 3 checkboxes to display Bold, Italic, & underline to the user */

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

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

```
class mycheckboxbox extends Frame implements ItemListener  
{
```



```

String msg = " ";
Checkbox c1, c2, c3;
Mycheckbox( )
{
    setLayout(new FlowLayout());
    c1 = new Checkbox("Bold" true);
    c2 = new Checkbox("Italic");
    c3 = new Checkbox("Underline");

    add(c1);
    add(c2);
    add(c3);
    c1.addItemListener(this);
    c2.addItemListener(this);
    c3.addItemListener(this);
    // close frame
    addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent we)
        {
            System.exit(0);
        }
    });
}
// this method is called when user clicks on Check Box
public void ItemStateChanged(ItemEvent ie)
{
    repaint(); // call paint() method
}
// display current state of checkboxes
public void paint(Graphics g)
{
    g.drawString("Current state:", 10, 100);
    msg = "Bold: " + c1.getState();
}

```

```

g.drawString(msg, 10, 120);
msg = " Italic : " + c2.getState();
g.drawString(msg, 10, 140);
msg = " Underline: " + c3.getState();
g.drawString(msg, 10, 160);
}
public static void main (String args[])
{
    Mycheckbox mc = new Mycheckbox();
    mc.setTitle("Mycheckbox");
    mc.setSize(400, 400);
    mc.setVisible(true);
}
}

```

o/p:

Mycheckbox		→ X
<input checked="" type="checkbox"/>	Bold	<input type="checkbox"/> Italic <input type="checkbox"/> Underline
Current state:		
Bold : true		
Italic : false		
Underline : false		

Java AWT canvas:

The Canvas Control represents a blank rectangular area where the ^{user} application can draw or trap input events from the user. It inherits the Component class.

// Program illustrating Canvas

```

import java.awt.*;
public class CanvasExample
{
    public CanvasExample()
    {

```

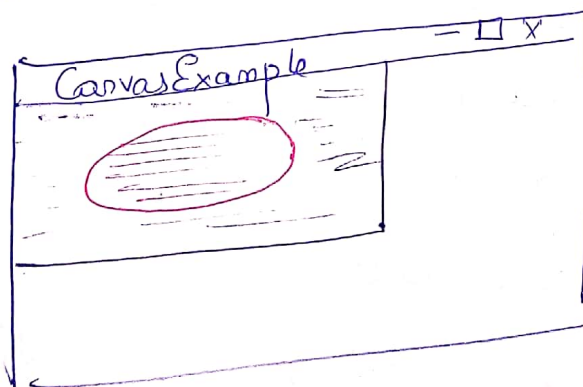


```

Frame f = new Frame("CanvasExample");
f.add(new MyCanvas());
f.setLayout(null);
f.setSize(400, 400);
f.setVisible(true);
}
public static void main(String args[])
{
    new CanvasExample();
}
}
class MyCanvas extends Canvas
{
    public MyCanvas()
    {
        setBackground(Color.GRAY);
        setSize(300, 200);
    }
    public void paint(Graphics g)
    {
        g.setColor(Color.red);
        g.fillOval(75, 75, 150, 75);
    }
}
}

```

o/p :



Radio Button :

- A radio button represents a round shaped button, such that only one can be selected from a group of buttons.
- Radio buttons can be created using `CheckboxGroup` class & checkbox classes.
- First, create a `CheckboxGroup` class object. While creating a radio button, we should pass `CheckboxGroup` object to the checkbox class. It represents the group to which the radio button belongs.
- When the same `Checkbox` object is passed to different radio buttons, then those radio buttons will be considered as belonging to same group & hence user is allowed to select only one from them.

→ To create a radio button, pass `CheckboxGroup` object to `Checkbox` class object

```
CheckboxGroup cbg = new CheckboxGroup();  
Checkbox cb = new Checkbox("label", cbg, state);
```

Here if the state is true then the radio button appears to be already selected by default. If the state is false, then the radio button appears normal as if it is not selected.

→ To know which radio button is selected by the user:

```
Checkbox cb = cbg.getSelectedCheckbox();
```

→ To know the selected radiobutton's label:

```
String label = cbg.getSelectedCheckbox().getLabel();
```


// Program demonstrating Radio Button Event

```
import java.awt.*;  
import java.awt.event.*;  
class Myradio extends Frame implements ItemListener
```

```
{  
    String msg=" ";  
    CheckboxGroup cbg;  
    Checkbox y, n;  
    Myradio()
```

```
{  
    setLayout(new FlowLayout());  
    cbg = new CheckboxGroup();  
    // create 2 radio buttons  
    y = new Checkbox("Yes", cbg, true);  
    n = new Checkbox("NO", cbg, false);
```

```
    add(y);
```

```
    add(n);
```

```
    y.addItemListener(this);
```

```
    n.addItemListener(this);
```

```
    addWindowListener(new WindowAdapter())
```

```
{  
    public void windowClosing(WindowEvent we)
```

```
{  
        System.exit(0);
```

```
    }
```

```
};
```

```
}  
public void itemStateChanged(ItemEvent ie)
```

```
{  
    repaint();
```

```
}  
public void paint(Graphics g)
```

```
{  
    msg = "Current selection: ";
```

```
    msg += cbg.getSelectedCheckbox().getLabel();
```

```
    g.drawString(msg, 10, 100);
```

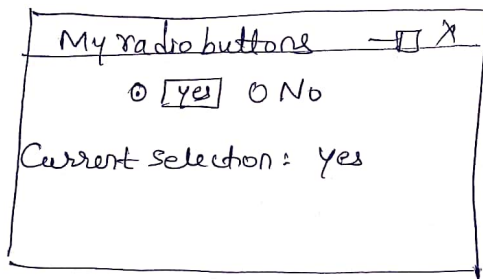
```
}
```

```

public static void main(String args[])
{
    Myradio mr = new Myradio();
    mr.setTitle("my radio buttons");
    mr.setSize(400, 400);
    mr.setVisible(true);
}

```

o/p:



• Grid Bag Layout Manager:

The java.awt.GridBagLayout manager is most powerful & flexible of all the layout managers but more complicated to use.

Unlike GridLayout where the components are arranged in a rectangular grid & each component in container is forced to be the same size, in GridBagLayout, components are also arranged in rectangular grid but can have different sizes & can occupy multiple rows & columns.

In order to create GridBagLayout, we first instantiate the GridBagLayout class by using its only no-arg constructor.

```
GridBagLayout layout = new GridBagLayout();
```

→ A GridBagLayout required a lot of information to know where to put a component in container.

A helper class called GridBagConstraints provides all this information. It specifies constraints on how to

position a component, how to distribute a component & how to resize & align them. Each component in a GridBagLayout has its own set of constraints, so you have to associate an object of type GridBagConstraints with each component before adding component to the container.

Field

Purpose

i) ~~the~~ gridx & gridy

These contains the coordinates of the origin of grid. They specify the x & y coordinate of the cell to which the component will be added. The default value is GridBagConstraints.RELATIVE. which indicates component can be added to the right of previous.

ii) gridwidth, gridheight

Specifies the height & width of component in terms of cells. The default is 1.

iii) weightx, weighty

Specifies a weight value that determines the horizontal & vertical spacing between cells & the edges of container that holds them. The default value is 0.0.

/* program demonstrating GridBagLayout */

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

```
class GridBagLayoutExample extends Frame
```

```
{
```

```
    GridBagLayoutExample()
```

```
{
```

```
    Label lblName = new Label("Name");
```

```
    TextField txtName = new TextField(10);
```

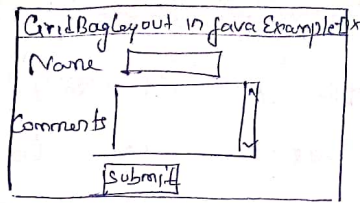


```

Label lblComments = new Label("Comments");
TextArea TAreaComments = new TextArea(6, 15);
Button btnSubmit = new Button("Submit");
setLayout(new GridBagLayout());
GridBagConstraints gc = new GridBagConstraints();
add(lblName, gc, 0, 0, 1, 1, 0, 0);
add(txtName, gc, 1, 0, 1, 1, 0, 20);
add(lblComments, gc, 0, 1, 1, 1, 0, 0);
add(TAreaComments, gc, 1, 1, 1, 1, 0, 60);
add(btnSubmit, gc, 0, 2, 2, 1, 0, 20);
}
void add(Component comp, GridBagConstraints gc, int x, int y,
        int w, int h, int wx, int wy)
{
    gc.gridx = x;
    gc.gridy = y;
    gc.gridwidth = w;
    gc.gridheight = h;
    gc.weightx = wx;
    gc.weighty = wy;
    add(comp, gc);
}
}
class GBLExample
{
    public static void main(String args[])
    {
        GridBagLayoutExample frame = new GridBagLayoutExample();
        frame.setTitle("Grid Bag Layout in Java Example");
        frame.setSize(300, 200);
        frame.setVisible(true);
    }
}

```

O/p:



Introduction to Swings:

→ The Abstract Window Toolkit (AWT) defines a basic set of controls, windows & dialog boxes that support a usable, but limited graphical interface. Because, AWT components internally depend on native methods like C functions & this is not desirable as C is system dependent language.

Therefore the look & feel of AWT components change depending on the platform (or operating system). For example, consider the code to create a push button in AWT. When this code is executed in windows, it will display windows-type of push button whereas the same code in UNIX will display UNIX style of push button i.e. its appearance changes from system to system.

→ Moreover, AWT components are heavy-weight. It means these components take more system resources like more memory & more processor time.

Due to these reasons AWT package is developed without internally taking the help of native methods. Hence, all the classes of AWT are extended to form new classes & new class library is created.

This library is called JFC (Java Foundation Class)

Java Swing

- Java Swing is a part of JFC that is used to create window based applications. It is built ~~as~~ on top of AWT API & entirely written in java.
- ~~The~~ Unlike AWT, Java Swing provides platform independent & lightweight components.

The java x.swing package provides classes for Java Swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckBox, JMenu etc.

Difference between AWT & Swing

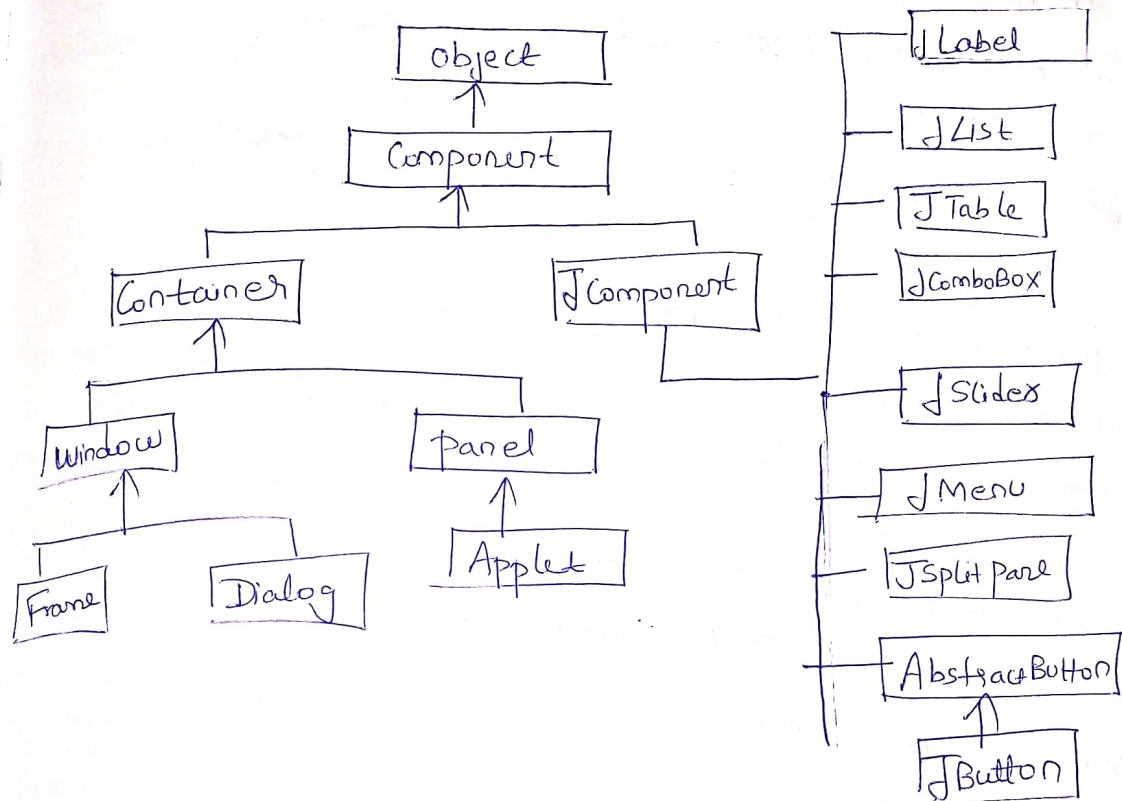
Java AWT

- ① AWT components are platform dependant. so there look & feel changes according to OS.
- ② AWT components are heavy weight
- ③ AWT provides less components than swing
- ④ AWT doesn't follow MVC (Model view Controller) where model represents data, view represents presentation & controller acts as interface between model & view

Java Swing

- ① Java Swing components are platform independent so there look & feel remains constant
- ② swing components are light weight
- ③ Swing provides more powerful components such as tables, lists, scrollpares, color choosers etc.
- ④ Swing follow MVC

The hierarchy of Java Swing API is given below.



/* program demonstrating JButton, JCheckBox, JRadio Button, JTextArea, JList classes in swings */

```

import javax.swing.*;
import javax.swing.awt.*;
import javax.swing.event.*;
public class testswing extends JFrame

```

```

{
    testswing ( )

```

```

{
    JButton bt1 = new JButton(new ImageIcon("D:\\car.jpg"));

```



```
JButton bt2 = new JButton("No");
JTextField jtf = new JTextField(20);
add(jtf);
JCheckBox jcb = new JCheckBox("yes");
add(jcb);
jcb = new JCheckBox("No");
add(jcb);
set jcb = new JCheckBox("maybe");
add(jcb);
setLayout(new FlowLayout());
setLayout(new FlowLayout());
setSize(400, 400);
add(bt1);
add(bt2);
JRadioButton jcb1 = new JRadioButton("A");
add(jcb1);
jcb1 = new JRadioButton("B");
add(jcb1);
jcb1 = new JRadioButton("C");
add(jcb1);
jcb1 = new JRadioButton("none");
add(jcb1);
setLayout(new FlowLayout());
```

```

setVisible(true);
JTextArea area = new JTextArea("welcome to
javaprogram");
area.setBounds(10, 30, 200, 200);
add(area);
DefaultListModel<String> l1 = new DefaultListModel<>();

```

```

// addElement("Item1");
// addElement("Item2");
// addElement("Item3");
// addElement("Item4");
JList<String> list = new JList<>(l1);

```

```

list.setBounds(100, 100, 75, 75);
add(list);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

```

```

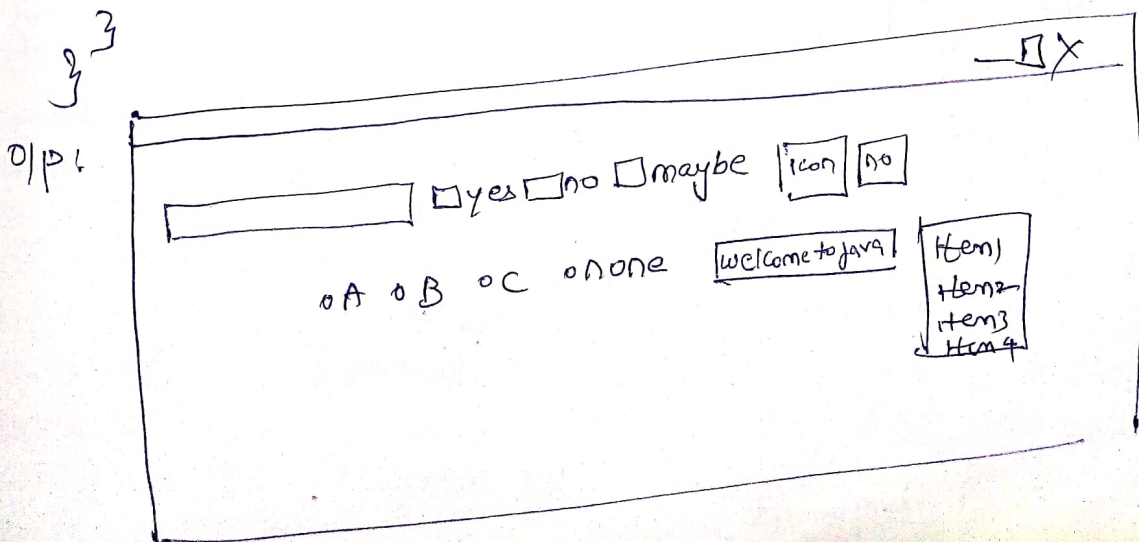
}
public static void main(String args[])

```

```

{
    new TestSwing();
}
}

```

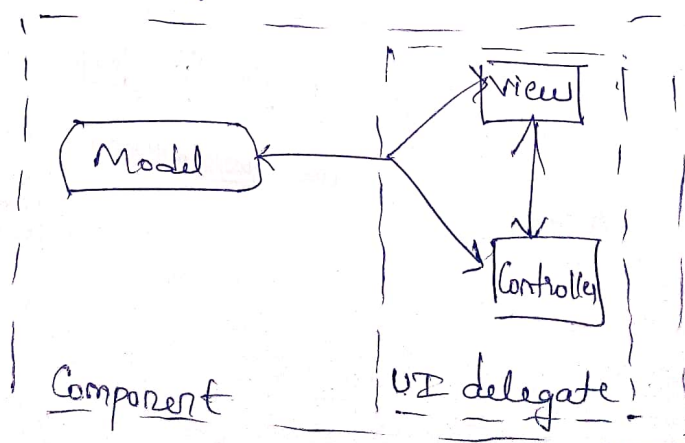


The Model view Controller Architecture :

- Swing uses the model-view controller architecture (MVC) as the fundamental design ^{behind} of each of its Component.
- Model corresponds to the state information associated with the component. For ex, in case of checkbox, the model contains a field that indicates if the box is checked or unchecked.
- The view determines how the component is displayed on the screen.
- The controller determines how the component reacts by ~~change~~ to the user. For example, when user clicks a checkbox, the controller reacts by changing the model to reflect the user's choice (checked or unchecked).

MVC in Swing.

Swing actually makes use of simplified variant of the MVC design called model-delegate. This design combines the view & the controller object into a single element that draws component to the screen & handles GUI events known as UI delegate.



Each Swing Component contains a model & UI delegate. The model is responsible for maintaining information about the component's state. The UI delegate is responsible for maintaining information about how to draw the component on the screen. In addition, UI delegate (in conjunction with AWT) reacts to various events that propagate through the component.

Swing Components & Containers

- A Component is an individual visual control. Swing Framework contains a large set of components which provide rich functionalities & allow high level customization. They all are derived from `JComponent` class.
- A Container holds a group of components. It provides space where a component can be managed & displayed. Containers are of two types.

① Top Level Containers

- It inherits `Component` & `Container` of AWT
- It cannot be contained within other containers
- Heavy weight

Example : `JFrame`, `JDialog`, `JApplet`

② Lightweight Containers.

- It ~~inherits~~ inherits `JComponent` class.
- It has general purpose containers
- It can be used to organize related components together

Ex : `JPanel`.

JButton

- JButton class provides functionality of a button.
- JButton has 3 Constructors:

- JButton(Icon ic)
- JButton(String str)
- JButton(String str, Icon ic) :

It allow a button to be created using Icon, a string or both. JButton supports ActionEvent. When a button is pressed ActionEvent is generated.

JTextField :

- JTextField is used for taking input of single line of text.

It has 3 Constructors.

- JTextField(int)
- JTextField(String str, int)
- JTextField(String str)

JCheckBox

- JCheckBox^{class} is used to create checkboxes in frame.

- JCheckBox(String str);

JRadioButton

RadioButton is group of related button in which only one ~~s~~ can be selected. JRadioButton is class is used to create radiobutton in Frames.

→ JRadioButton(String str)

→ JRadioButton(String s, boolean selected) ÷
Creates a radio button with specified text & selected status.

JList

The object of ~~JList~~ JList class represents a list of text items. The list of text items can be set up so that user can choose either one item or multiple items. It inherits JComponent class.

Commonly used Constructors:

- JList() : creates a list with an empty, read only ^{model} ~~model~~
- JList(array[] list Data) : creates a JList that displays the elements in specified array.
- JList(ListModel <array> data model) : creates a JList that displays elements from the specified, non-null, model

Note: API provides a default implementation of this class named DefaultListModel.

Java Swing Examples:

There are two ways to create a Frame:

- By creating the object of Frame class (association)
- By extending Frame class (inheritance)

Ex:
/x creating a button & adding it on frame object inside the main() method x/

```
import javax.swing.*;
```

```
class test1
```

```
{  
    public static void main (String args[])
```

```
{  
    JFrame f = new JFrame(); // creating instance of Frame
```

```
    JButton b = new JButton("Click");
```

```
    b.setBounds (130, 100, 100, 40);
```

```
    f.add (b);
```

```
    f.setSize (400, 500);
```

```
    f.setLayout (null);
```

```
    f.setVisible (true);
```

```
}
```

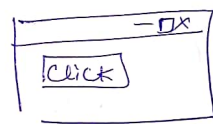
/x Swing by Association inside Constructor x/

```
import javax.swing.*;
```

```
public class sim test2
```

```
{
```

```
    JFrame f;
```



ame :
(association)

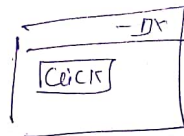
e object

f Frame

or x

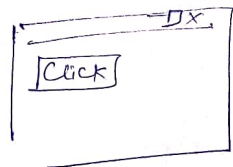
~~Star~~
test2()

```
{
    f = new JFrame();
    JButton b = new JButton("Click");
    b.setBounds(130, 100, 100, 40);
    f.add(b);
    f.setSize(400, 500);
    f.setLayout(null);
    f.setVisible(true);
}
```



→ We can also inherit JFrame class, so there is no need to create the instance of JFrame explicitly
import java.x.swing.*; ^{by inheritance}
class test3 extends JFrame

```
{
    test3()
    {
        JButton b = new JButton("Click");
        b.setBounds(130, 100, 100, 40);
        add(b);
        setSize(400, 500);
        setLayout(null);
        setVisible(true);
    }
    public static void main(String args[])
    {
        new test3();
    }
}
```



Note:

- getDefaultCloseOperation (Constant) method of JFrame

Class is used to close the frame.

Where the constant is:

JFrame.EXIT_ON_CLOSE - This closes application upon clicking on close button.

/* program to terminate application by clicking on close button of frame */

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class testswing extends JFrame
{
    testswing()
    {
        JButton b1 = new JButton("yes");
        JButton b2 = new JButton("NO");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        add(b1);
        add(b2);
        setSize(400, 500);
        setLayout(null);
        setVisible(true);
        setLayout(new FlowLayout());
    }
}
```


f JFrame

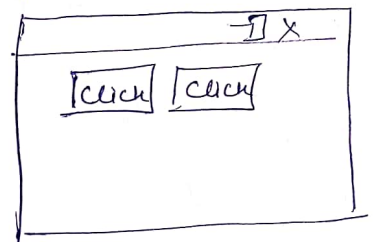
Location

Working on

E);

```
public static void main (String[] args)
```

```
{  
    new test4 ();  
}  
}
```



Introduction

Introduction to Networking in Java

- Java Networking is a concept of connecting two or more computing devices together. So that we can share resources.
- Java socket programming provides facility to shared data between different computing devices.

Advantage of Java Networking

- ① Sharing resources
- ② Centralize Software management

The widely used Java Networking terminologies:

- ① IP Address
- ② protocol
- ③ port Number
- ④ MAC Address
- ⑤ Connection oriented & Connectionless protocol
- ⑥ Socket

1) IP Address :

The identifier used to identify each device connected to internet is called internet address or IP Address.

The two IP addressing standards we use are:

IPv4 & IPv6

IPv4 address consists of 4 bytes (32 bits) also known as octets. While IPv6 address are 16 bytes (128 bits) long i.e. IP Address contains some bytes which identify the network & the actual computers inside the network.

Ex: 87.248.113.14

This IP address may represent, for ex a website on server machine on internet as www.yahoo.com

→ It contains 4 integer numbers in range of 0 to 255 & separated by dot.

② protocol:

- A protocol represents set of rules to be followed by every computer on network to physically move data from one place to another place on a network.

TCP (Transmission Control protocol) / IP (Internet protocol) is the standard protocol model used on any network including internet. & some other protocols such as FTP, SMTP, HTTP, Telnet, POP.

③ Port Number.

The port Number is used to uniquely identify different applications. It acts a communication end point between applications. The port Number is associated with the IP Address from communication between two applications.

④ MAC Address (Medium Access control): It is unique identifier of Network Interface Controller.

⑤ Socket ÷ Socket is a point of connection between a server & client on a network. Each socket is given an identification number, which is called 'port number'. Port number takes 2 bytes & can be of form 0 to 65535 used to identify socket uniquely.

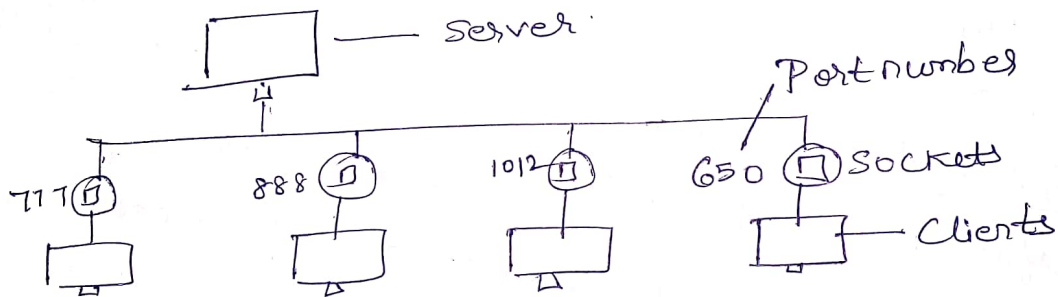
Establishing communication between a server & a client using socket is called 'socket programming'.

→ We should use new port number for each new socket &

Fig: Already ~~all~~ allotted port numbers for the services are shown below:

Port number	Application or service
13	Data & time services
21	FTP (File transfer protocol) which transfers file
23 23	Telnet, which provides remote login
25	SMTP, which delivers mail
80	HTTP (Hypertext transfer protocol) which transfers web page

Fig: A server connected with clients using sockets



⑥ Connection oriented & Connectionless protocol:

- In connection oriented protocol, acknowledgement is sent by receiver. So it is reliable but slow. Ex is TCP.
- In Connectionless protocol, ack is not sent by receiver. So it is not reliable but fast. Ex is UDP.

Java Inet Address class

Java Inet Address class represents IP Address. The java.net Inet Address class provides methods to get the IP of any host name. for ex www.google.com

Commonly used methods of Inet Address:

Method

1) Public static Inet Address
getByName(String host) throws
Unknown Host Exception

Description
→ it returns the instance of
Inet Address containing
localhost IP & name

2) public static Inet Address
getLocalHost() throws
Unknown Host Exception

→ it returns the instance of
Inet Address containing local host
name & address.

3) public String getHostName()

→ It returns the host name
of IP & address

4) public String getHostAddress()

→ It returns IP Address in
String format.

/* program illustrating Inet Address class to get ip address
of ^{website} ~~www.google.com~~ */

```
import java.io.*;  
import java.net.*;
```

```
class InetDemo
```

```
{
```



```

public static void main (String[] args)
{
    BufferedReader br = new BufferedReader(new
        InputStreamReader (System.in));

    try
    {
        System.out.println (" enter a website name: ");
        String site = br.readLine ();
        InetAddress ip = InetAddress.getBy Name (site);
        System.out.println ("Host Name: " + ip.getHost Name());
        System.out.println ("IP Address: " + ip.getHost Address());
        System.out.println ("IP localhost Address: " + ip.getLocal Host());
    }
    catch (Exception e)
    {
        System.out.println (e);
    }
}

```

O/P: c:\>javac InetDemo.java

c:\>java InetDemo

enter a website name

www.google.com

Host Name: www.google.com

IP Address: 216.58.196.164

localhost IP Address: swapna/162.22.23.251

Java Socket Programming

- Java Socket Programming is used for communications between the applications running on different JRE.
- Java Socket Programming can be connection oriented or connection less.
- Socket & ServerSocket classes are used for connection oriented socket programming & DatagramSocket & DatagramPacket classes are used for connection less socket programming.

The client in socket programming, must know two information:

- ① IP Address of server, &
- ② port number.

Socket Class

A socket is simply an endpoint for communications between the machines. The socket class can be used to create a socket.

Important methods:

<u>Method</u>	<u>Description</u>
① public InputStream getInputStream()	→ Returns the InputStream attached with this socket
② public OutputStream getOutputStream()	→ Returns the output stream attached with this socket
③ public synchronized void close()	→ Close this socket

Server Socket class

The Server Socket class can be used to create a server socket. This object is used to establish connection with clients.

Important methods

Description

① public Socket accept()

- returns socket & establish a connection between server & client

② public Synchronized void close() - closes the server socket.

Example of socket programming in which client sends a text & server receives it

File: Myserver.java

```
import java.io.*;
import java.net.*;
public class Myserver
{
    try
```

```
{
    ServerSocket ss = new ServerSocket(6666);
```

```
    Socket s = ss.accept(); // establishes connection
```

```
    DataInputStream dis = new DataInputStream(s.getInputStream());
```

```
    String str = (String)dis.readUTF();
```



```

        System.out.println("message()=" + str);
        ss.close();
    }
    catch (Exception e)
    {
        System.out.println(e);
    }
}
}
}

```

Note:

→ The `java.io.DataInputStream.readUTF()` method reads in a string that has been encoded using a modified UTF-8 format. The string of character is decoded from the UTF & returned as string

File: MyClient.java

```

import java.io.*;
import java.net.*;

class myclient
{
    public static void main(String args[])
    {
        try
        {
            Socket s = new Socket("localhost", 6666);
            DataOutputStream dout = new DataOutputStream(s.getOutputStream());

            dout.writeUTF("Hello Server");
            dout.flush();
            dout.close();
            s.close();
        }
    }
}

```



```

catch (Exception e)
{
    System.out.println(e);
}
}
}

```

Note:

The `java.io.DataOutputStream.writeUTF(String str)` write a string to the underlying output stream using modified UTF-8 encoding

```
public final void writeUTF(String str)
```

→ `str` - a string to be written to the output stream

- To execute this program open two Command prompts & execute each program at each Command prompt as displayed below:

After running the client application, a message will be displayed on the server console.

```

d:\> javac client.java
d:\> java myclient
d:\>

```

```

d:\> javac myserver.java
d:\> java Myserver
message = Hello server

```

Java URL class

The Java URL class represents an URL. URL is acronym for Uniform Resource Locator. It points to a resource on world wide web. For example `http://www.geeksforguns.org/index.html`
~~`http://www.facebook.com`~~

The URL contains 4 parts ÷

- ① protocol ÷ In this case, http is protocol
- ② Server name or Ip Address ÷ In this case, www.geeksforguns.org is the server name.
- ③ port Number ÷ It is optional attribute
- ④ File Name or directory name ÷ It is the file that is referred. In the above example it is index.html.

Commonly used methods of Java URL class

<u>Method</u>	<u>Description</u>
i) <code>Public String getProtocol()</code>	→ It returns the protocol of URL
ii) <code>public String getHost()</code>	→ It returns the host name of URL
iii) <code>public String getPort()</code>	→ It returns the PortNumber of URL
iv) <code>public String getFile()</code>	→ It returns the file name of URL
v) <code>public URLConnection openConnection()</code>	→ It returns returns the instance of URLConnection is associated with this URL

/* program to receive different parts of a URL & supplied to URL class object */

```
import java.net.*;
class URLDemo
{
    public static void main (String args[])
    {
        try
        {
            URL url = new URL("http://www.google.com/abc.html");
            System.out.println("Protocol:" + url.getProtocol());
            System.out.println("Host Name:" + url.getHost());
            System.out.println("Port Number:" + url.getPort());
            System.out.println("File Name:" + url.getFile());
            System.out.println("External form:" + url.toExternalForm());
        }
        catch (Exception e)
        {
            System.out.println(e);
        }
    }
}
```

o/p :

Protocol: http

Host Name: www.google.com

Port Number: -1

Filename: /abc.html

External form: http://www.google.com/abc.html

Inheritance

Inheritance can be defined as the process where one class acquires the properties of (methods & fields) of another.

The class which inherits the properties of other is known as subclass (derived class, child class).

The class whose properties are inherited is known as Super class (base class, parent class).

→ Inheritance can be useful when two or more classes have some fields & operations in common. Instead of duplicating these common aspects in various classes, we can keep them in base class, that other class inherit from it.

→ To inherit a class, incorporate the definition of one class into another by using extends keyword.

Syntax of extends keyword (General form of class declaration that inherits superclass)

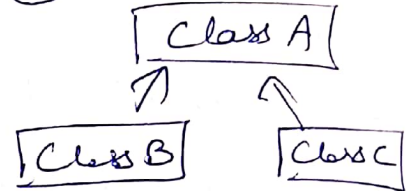
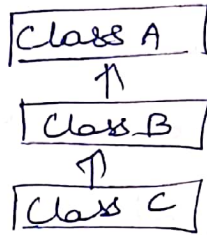
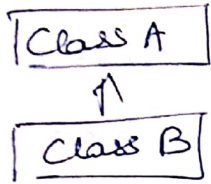
```
{
-
}
class subclass_name extends superclassname
{
-
} // body of class
```

• Inheritance represents IS-A relationship which is also known as parent-child relationship.

→ Types of inheritance:

on the basis of class, there ~~can~~^{can} be 3 types of inheritance: single, multilevel & hierarchical
- In java programming, multiple & hybrid inheritance is supported through interface only.

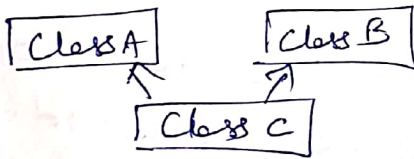
(i) Single inheritance (ii) Multilevel inheritance (iii) Hierarchical inheritance



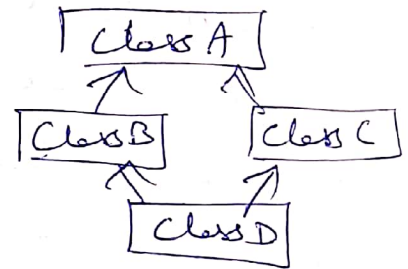
Note: Multiple inheritance is not supported in java through class.

When one class inherits multiple classes, it is known as

④ Multiple inheritance



⑤ Hybrid inheritance



// Program to demonstrate inheritance

// create a superclass

class A

{

int i, j;

void showij()

{

System.out.println("i and j: " + i + " " + j);

}

}

// Create a subclass by extending class A
class B extends A

```
{ int k;  
  void showK()  
{  
    System.out.println("k: " + k);  
}  
}  
void sum()  
{  
    System.out.println("i+j+k: " + (i+j+k));  
}  
}
```

class SimpleInheritance

```
{ public static void main (String args[])
```

```
{  
    A superob = new AC();
```

```
    B subob = new BC();
```

// The superclass may be used by itself

```
superob.i = 10;
```

```
superob.j = 20;
```

```
System.out.println("Contents of superob");
```

```
superob.showj();
```

```
System.out.println();
```

/* The subclasses has ~~to~~ access to all public members of

its superclass */

```
subob.i = 7;
```

```
subob.j = 8;
```

```
subob.k = 9;
```



```

System.out.println(" Contents of subob: ");
    subob.showj();
    subob.showk();
    System.out.println();
    System.out.println(" Sum of i, j and k in subob: ");
    subob.sum();
}
}

```

3 }

o/p: Contents of superob:

i and j : 10 20

Contents of subob:

i and j : 7 8

k : 9

Sum of i, j and k in subob:

i + j + k : 24

Modifiers:

Public ÷ It allows method & data members to be accessed from any where, from the same class or different class

Private ÷ Any data member or method which is private can be accessed only from its class. Accessing it from different class will give error.

Ex: class car

```

{
    private void print()

```

```

{
    System.out.println("This is private method");
}
}

```

class Bike extends car

```

{
    public void printb()

```

```

{

```

```

        System.out.println("This is public method");
    }
}

```

class test

```

{
    public static void main (String args[])

```

```

{
    Bike b1 = new Bike();
    b1.print();
}
}

```

o/p = Cannot find Symbol
b1.print();

// program demonstrating single inheritance

class base

```

{
    int a=10, b=20;

```

```

    public void showab()

```

```

{
    System.out.println("a=" + a + " b=" + b);
}
}

```

class derived1 extends base

```

{
    int c=30, d=40;

```

```

    public void showc()

```

```

{
    System.out.println("c=" + c + " d=" + d);
}
}

```

```

    public void showall()

```

```

{
    System.out.println("a=" + a + " b=" + b + " c=" + c + " d=" + d);
}
}

```

Class derived

```
{  
    public static void main (String args[])
```

```
{  
    derived dr = new derived();
```

```
    dr.showabc();
```

```
    dr.showcd();
```

```
    dr.showall();
```

```
}
```

```
}
```

O/p: a=10 b=20

c=30 d=40

a=10 b=20 c=30 d=40

→ Protected : It only allows methods or data members to be access from its class & its subclass

Ex: Class vehicle

```
{  
    protected String brand = "maruti";
```

```
    public void display()
```

```
{
```

```
        System.out.println("Hello");
```

```
}
```

```
}
```

Class car extends vehicle

```
{
```

```
    private String modelname = "suzuki";
```



```

public static void main (String args[])
{
    Car mycar = new car();
    mycar.display();
    System.out.println(mycar.brand + " " +
        mycar.modelname);
}
}

```

o/p: Hello
maruti suzuki

Access modifiers in Java

The access modifiers specifies accessibility (scope) of data member method, constructor or class

There are 4 types of java access modifiers:

① private ② default ③ protected ④ public

private access modifier : It is accessible only within class

Ex: class A

```

{
    private int data=40;
    private void msg()
    {
        System.out.println("Hello java");
    }
}

```

Public class simple

```

{
    public void static main (String args[])
    {

```

```

        A obj = new A();

```

```

        System.out.println (obj.data); // Compile time error

```

```

    } } obj.msg(); // Compile time error

```

→ The Super keyword.

Super is a reference variable that is used to refer immediate parent class object

Following are scenarios where super keyword is used:

- It is used to differentiate the members of superclass from members of subclass, if they have same names.
- Super() is used to invoke immediate parent class constructor
- super is used to invoke parent class method & parent class variable.

* Program illustrating how super can be used to access parent class method *

```
class Super1
{
    int i = 10;
    void m1()
    {
        System.out.println("Super class method");
    }
}
class Child2 extends Super1
{
    int i = 20;
    void m1()
    {
        System.out.println("value of i" + super.i);
        super.m1();
    }
}
```

```

class SuperTest
{
    public static void main (String args[])
    {
        Child2 obj = new Child2();
        obj.m1();
    }
}

```

o/p → value of i 10
superclass method

/* program demonstrating super() used to invoke
parent class constructor */

class test

```

{
    int a = 10;

```

```

    test(int x)
    {

```

```

        System.out.println("Test can display..");
        System.out.println("The value of x : " + x);
    }

```

```

    void disp()
    {

```

```

        System.out.println("--The parent class--");
        System.out.println("The value a : " + a);
    }
}

```

```

class Display extends test
{

```

```

    {

```

```

        int a = 20;
    }
}

```



```
Display (int y)
```

```
{  
    super(y); // super class constructor call  
    System.out.println("-- Display can display --");  
    System.out.println("The value y " + y);  
}
```

```
void dis()
```

```
{  
    super.dis(); // super class method call  
    System.out.println("-- child class --");  
    System.out.println("The value a : " + a);  
}
```

```
class demosuper
```

```
{  
    public static void main(String args[])
```

```
{  
    Display d1 = new Display(123);
```

```
    d1.dis(); // method call.
```

```
}
```

o/p: Test car display

The val x : 123

-- Display can display --

The val y : 123

-- parent class --

The val a : 10

-- child class --

The value : 20

Note: A Subclass inherits all the members (fields, methods & nested classes) from its superclass.

Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from subclass.

/* Subclass with constructor */
class Rectangle

```
{  
    protected int length;  
    protected int breadth;  
    public Rectangle (int l, int b)
```

```
{  
    length = l;  
    breadth = b;
```

```
}  
    public int getArea()
```

```
{  
    return length * breadth;
```

```
}  
    public int getParameter()
```

```
{  
    return 2 * (length + breadth);
```

```
}  
}
```

class square extends Rectangle

```
{  
    public square (int a)
```

```
{  
    super (a, a);
```

```
}  
}
```

class sub

```
{
    public static void main (String args[])
    {
        Square s = new square(2);
        int ar, p;
        ar = s.getArea();
        p = s.getPerimeter();
        System.out.println("Area is " + ar);
        System.out.println("Perimeter is " + p);
    }
}
```

O/p:
Area is 4
Perimeter is 8

Note: super() is added in each class constructor automatically by compiler at beginning of the constructor of each class.

Ex:-

```
class Rectangle
{
    public Rectangle()
    {
        System.out.println("This is constructor of parent class");
    }
}
class square extends Rectangle
{
    public square()
    {
        System.out.println("This is constructor of child class");
    }
}
```



```

public void run()
{
    System.out.println("This is method of child class");
}
}
class sub
{
    public static void main (String args[])
    {
        Square s = new Square();
        s.run();
    }
}

```

Q11: This is constructor of parent class
 " " " of child "
 " " method of " "

→ Final Keyword in Java

The final keyword in Java is used to restrict the user. It can be used in 3 different ways.

Final can be
 ① variable ② method ③ class.

① Final variable ÷ If any variable is made as final, the value of it cannot be changed i.e. we can't change the value of final variable once it is initialized.

Ex: // Program demonstrating final variable

Class demo

```
{
```

```
final int max_value = 99;
```

```
void mymethod()
```

```
{  
    max_value = 101;
```

```
}
```

```
public static void main (String args[])
```

```
{  
    demo obj = new demo();
```

```
    obj.mymethod();
```

```
}  
}
```

O/p: Error: Cannot assign a value to final variable
max_value=101

→ Blank final variable.

A final variable that is not initialized at the time of declaration is known as blank final variable. ~~Blank~~ Blank final variable must be initialized in constructor of the class ~~otherwise~~ it will throw compilation error.

```
class demo
```

```
{
```

```
    // blank final variable
```

```
    final int max_value is;
```

```
    demo()
```

```
{
```

```

// blank final variable
final int max-value;
demo()
{ // it must be initialized in constructor
  max-value = 100;
}
void mymethod()
{ System.out.println(max-value);
}
public static void main(String args[])
{ demo obj = new demo();
  obj.mymethod();
}

```

What is the use of blank final variable
 Let's say we have a student class which is having a field called RollNo. Since RollNo should not be changed ~~to~~ once the student is registered, we can declare it as a final variable in a class but we can't initialize rollno in advance for all the students (otherwise all students would be having same rollno). In such case we can declare rollno variable as blank final & we initialize this value during object creation as shown in given example:

Class studentdata

```
{ // blank final variable
```

```
final int rollno;
```

```
studentdata(int rnum)
```

```
{ // it must be initialized in constructor
```

```
rollno = rnum;
```

```
}
```

```
void mymethod()
```

```
{ System.out.println("roll no is " + rollno);
```

```
}
```

```
public static void main(String args[])
```

```
{ studentdata obj = new studentdata(1234);
```

```
obj.mymethod();
```

```
}
```

O/p : rollno = 1234

→ Static blank final variable:

A static blank final variable that is not initialized at time of declaration is known as static blank final variable. It can be initialized only in static block.

Class A

```
{ static final int data; // static block final variable
```

static

```
{ data = 50;
```

```
}
```

```
public static void main (String args[])
```

```
{ System.out.println(A.data); // o/p: 50
```

```
}
```

```
}
```

→ if parameter is declared as final, ^{we} ~~it~~ can't be changed value of it

class bike)

```
{ int cube(final int n)
```

```
{ n = n + 3; // can't be changed as n is final
```

```
  A = n * n * n;
```

```
}
```

```
public static void main (String args[])
```

```
{
```

```
  bike b = new bike();
```

```
  b.cube(5);
```

```
} }
```

O/p:
error:

Final method:

A final method can't be overridden. which means even though a subclass can call the final method of parent class, but can't override it.

Ex: class xyz

{
final void demo()

{
System.out.println("xyz class method");

}

class ABC extends xyz

{
void demo()

{
System.out.println("ABC class method");

}

public static void main(String args[])

{
ABC obj = new ABC();

obj.demo();
}

}
op: error: demo() in ABC can't override demo() in xyz

Final class: Class which cannot be inherited
is final class.

~~final class xyz~~

Ex:

Q: // Program demonstrating use of final class & final method

```
final class finaldemo
```

```
{ final void mic() }
```

```
{ System.out.println("parent class"); }
```

```
}  
class finalchild extends finaldemo
```

```
{ void mic() }
```

```
{ System.out.println("child class"); }
```

```
}
```

```
class finaltest
```

```
{ public static void main(String args[])
```

```
{ finalchild obj = new finalchild();  
obj.mic(); }
```

```
}
```

error: cannot inherit from final demo class finalchild
extends finaldemo.
mic() in finalchild can't override mic() in finaldemo

9/1/19

Abstract methods & classes:

→ Runtime polymorphism or Dynamic Method Dispatch

→ Polymorphism in java is a concept by which we can perform a single action in different ways.

There are two types of polymorphism in java -

- Compile time polymorphism &
- Run time polymorphism.

- we can perform polymorphism in java by method overloading & method overriding
- If you overload a static method in java, it is example of compile-time polymorphism

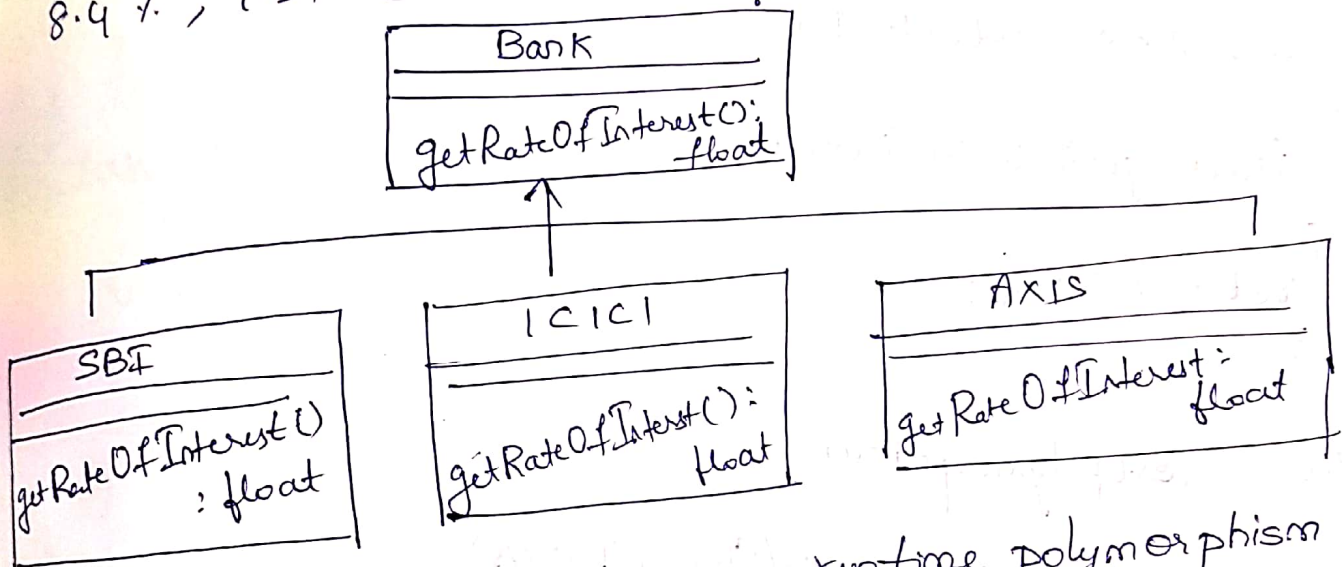
⇒ Runtime polymorphism or Dynamic method Dispatch is a process in which a call to an overrider method is resolved at runtime rather than compile-time

• In this process, an overrider method is called through reference variable of superclass. The determination of the method to be called is based on object being referred to by a reference variable

upcasting - When a parent class reference variable refers to a child class object, it is known as upcasting

Java Run-time polymorphism Example: Bank.

Consider a scenario where Bank is a class that provides a method to get rate of interest. However, the rate of interest may differ according to bank. However, the rate of interest may differ according to banks. for ex: SBI, ICICI & AXIS banks are providing 8.4%, 7.3% & 9.7% rate of interest.



// program demonstrating runtime polymorphism

```
class Bank
```

```
{
    float getRateOfInterest()
{
```

```
    return 0;
}
```

```
}
class SBI extends Bank
```

```
{
    float getRateOfInterest()
{
```

```
    return 8.4f;
} }
```


Class ICICI extends Bank

```
{  
    float getRateOfInterest()  
    {  
        return 9.7f;  
    }  
}
```

class AXIS extends Bank

```
{  
    float getRateOfInterest()  
    {  
        return 7.3f;  
    }  
}
```

Class Testpolymorphism

```
{  
    public static void main(String[] args)  
    {  
        Bank b;  
        b = new SBIC();  
        System.out.println("SBI rate of interest: " + b.getRateOfInterest());  
        b = new ICICIC();  
        System.out.println("ICICI rate of interest: " + b.getRateOfInterest());  
        b = new AXIS();  
        System.out.println("AXIS rate of interest: " + b.getRateOfInterest());  
    }  
}
```

3 O/P : SBI Rate of interest : 8.4

ICICI " " 9.7

AXIS " " 7.3

// runtime polymorphism (method overriding) with
multilevel inheritance */
class profession

```
{ void checkC )
```

```
{ System.out.println (" profession check");
```

```
}
```

```
} class Employee extends profession
```

```
{ void checkC )
```

```
{ System.out.println (" Employee check");
```

```
}
```

```
} class Teacher extends Employee
```

```
{ void checkC )
```

```
{ System.out.println ("Teacher check");
```

```
}
```

```
public static void main (String args [])
```

```
{ profession obj1, obj2, obj3;
```

```
obj1 = new professionC );
```

```
obj2 = new EmployeeC );
```

```
obj3 = new TeacherC );
```

```
obj1 . checkC );
```

```
obj2 . checkC );
```

```
obj3 . checkC );
```

```
} }
```

O/p: profession class
Employee "
Teacher "

→ Java Runtime polymorphism with data member,
- A method is overriden, not the data members,
So runtime polymorphism can't be achieved by data members.

```
Ex: class bike
{
    int speedlimit=90;
}
class honda extends bike
{
    int speedlimit=150;
    public static void main(String args[])
    {
        Bike obj=new honda();
        System.out.println(obj.speedlimit);
    }
}
```

o/p: 90

*/Design a vehicle class hierarchy in java & develop a program to demonstrate polymorphism */

```
class vehicle
{
    String regno;
    int model;
    vehicle (String r, int m)
    {
        regno=r;
        model=m;
    }
}
```



```
void display( )
```

```
{ System.out.println ("Registration no: " + regno);  
  System.out.println ("Model no: " + model);
```

```
}
```

```
} class TwoWheeler extends vehicle
```

```
{ int noOfWheel;
```

```
TwoWheeler (String r, int m, int n)
```

```
{ Super(r, m);  
  noOfWheel = n;
```

en-

ses.

```
} void display( )
```

```
{ System.out.println ("Two wheelers are");  
  Super.display();
```

```
System.out.println ("No. of wheel: " + noOfWheel);
```

```
} } class ThreeWheeler extends vehicle
```

```
{ int noOfWheel;
```

```
ThreeWheeler (String r, int m, int n)
```

```
{ Super(r, m);  
  noOfWheel = n;
```

```
}
```

```

void display()
{
    System.out.println("Three wheeler auto");
    Super.display();
    System.out.println("No of wheel: " + noofwheel);
}
}

Class Fourwheeler extends vehicle
{
    int noofwheel;
    Fourwheeler (String r, int m, int n)
    {
        super(r, m);
        noofwheel = n;
    }
    void display()
    {
        System.out.println("four wheeler car");
        Super.display();
        System.out.println("No. of wheel: " + noofwheel);
    }
}

class vehicleDemo
{
    public static void main (String args[])
    {
        Twowheeler t1;
        Threewheeler th1;
        Fourwheeler f1;
        t1 = new Twowheeler("TN 74 12345", 1, 2);
        th1 = new Threewheeler("TN 74 54321", 4, 3);
        f1 = new Fourwheeler("TN 34 45677", 5, 4);
        t1.display();
        th1.display();
        f1.display();
    }
}

```

Q11: Two wheeler tvs
Registration no: TN74 12345
Model no: 1
No. of wheel: 2
Three wheeler auto
Registration no: TN74 54321
Model no: 4
No. of wheel: 3
Four wheeler car
Registration no: TN34 45677
Model no: 5
No. of wheel: 4

→ Abstract methods & classes :


Abstraction is a process of hiding the implementation details & showing only functionality to the user.
There are two ways to achieve abstraction in java
① Abstract class ② Interface.

→ Abstract class :

- A class that is declared using "abstract" keyword is known as abstract class. It can have abstract methods (methods without body) as well as concrete methods (regular methods with body).
A normal class (non abstract class) cannot have abstract methods.

→ Abstract method

- A method which is declared as abstract & doesn't have any implementation is known as abstract method.

Rules:  Example of Abstract class - that has
a abstract method. In this example, Bike is
an abstract class that contains only one abstract method
run. Its implementation is provided by Honda class

```
abstract class bike
{
    abstract void run();
}
class Honda extends bike
{
    void run()
    {
        System.out.println("running safely");
    }
    public static void main (String args[])
    {
        Bike obj = new Honda();
        obj.run();
    }
}
```

O/P: running safely.

Rules :

Note:

① Abstract class can't be instantiated which means we can't create the object of it. To use this, we need to create another class that extends this class & provides the implementation of abstract methods, then we can use object of that child class to call non abstract methods of parent class as well as implemented methods (those that were abstract in parent but implemented in child class).

- ② A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.
- ③ If a child class does not implement all the abstract methods of abstract parent class, then the child class must need to be declared abstract as well.
- Since abstract class allows concrete methods as well, it doesn't provide 100% abstraction.
- ④ An abstract class must be extended & in a same way abstract method must be overridden.
The class which is extending abstract class must override all the abstract methods.
- ⑤ An abstract class can have a data member, abstract method, non abstract method, constructor & even main method.

Why we need an abstract class?

Lets say we have a class Animal that has a method sound() & the subclasses of it like Dog, cow, Lion, cat etc. Thus, making this method abstract would be good choice, as by making this method abstract we force all the subclasses to implement this method (otherwise we get compilation errors), also we need not give any implementation in parent class. This way we ensures that every animal has a sound.

// Abstract class Example

```
abstract class Animal
{
    public abstract void sound();
}
class dog extends Animal
{
    public void sound()
    {
        println("woof");
    }
}
class cow extends Animal
{
    public void sound()
    {
        println("blah");
    }
}
public static void main (String args[])
{
    Animal obj;
    dog d = new dog();
    Cow c = new cow();
    obj = d;
    obj.sound();
    obj = c;
    obj.sound();
}
```

O/p: woof
 blah

Ex: // program demonstrating all the objects need
different implementations of same method.

abstract class MyClass

```
{
    abstract void calculate (double x);
}
```



```
class sub1 extends MyClass
```

```
{ // calculate Square value
```

```
void calculate(double x)
```

```
{ System.out.println("Square=" + (x * x));
```

```
}
```

```
class sub2 extends MyClass
```

```
{ // calculate Square root value
```

```
void calculate(double x)
```

```
{ System.out.println("Square root=" + Math.sqrt(x));
```

```
}
```

```
class sub3 extends MyClass
```

```
{ // calculate cube value
```

```
void calculate(double x)
```

```
{ System.out.println("Cube=" + (x * x * x));
```

```
}
```

```
class Different
```

```
{
```

```
public static void main(String args[])
```

```
{
```

```
sub1 obj1 = new sub1();
```

```
sub2 obj2 = new sub2();
```

```
sub3 obj3 = new sub3();
```

```
obj1.calculate(3);
```

```
obj2.calculate(4);
```

```
obj3.calculate(5);
```

```
}
```

O/P:

Square = 9.0

Square root = 2.0

Cube = 125.0

* Program calculating electricity bill for commercial & domestic plans using abstract class */

abstract class plan

{

protected double rate;

public abstract void getRate();

public void calculateBill (int units)

{

System.out.println ("bill amount for " + units + " units: ");

System.out.println (rate * units);

}

class commercialplan extends plan

{ public void getRate()

{ rate = 5.00;

}

}

class domesticplan extends plan

{ public void getRate()

{ rate = 2.60;

}

class calculate

{ public static void main (String args [])

{ plan p;

Commercialplan c = new Commercialplan();

domesticplan d = new domesticplan();

```
System.out.println (" commercial connection : ");
```

```
P = C;
```

```
P.getRate();
```

```
P.calculateBill (250);
```

```
System.out.println (" domestic connection ");
```

```
P = d;
```

```
P.getRate();
```

```
P.calculateBill (150);
```

O/P:

Commercial connection:

bill amount for 250 units : 1250.0

domestic connection

bill amount for 150 units : 370.0

```
}  
}
```

Example of abstract class that has constructor & abstract method */
abstract class Bike

```
{ Bike() }
```

```
{ System.out.println ("bike is created"); }
```

```
abstract void run();
```

```
void changeGear();
```

```
{ System.out.println ("gear changed"); }
```

```
}  
}
```

```
class Honda extends Bike
```

```
{ void run() }
```

```
{ System.out.println ("running safely"); }
```

```
}  
}
```


Class Test Abstraction

```
{  
    public static void main(String args[])  
    {  
        Bike obj = new Honda();  
        obj.run();  
        obj.changeGear();  
    }  
}
```

o/p : bike is created
running safely..
gear changed.

19/1/19

Managing Errors & Exceptions : Exception Handling

What is an Exception?

Exception is an event that occurs interrupts the normal flow of execution. It is a disruption during the execution of java program.

→ Types of errors : There are basically 3 types of errors in java

(i) Compile time errors : These errors are errors which prevents the code from compiling because of error in Syntax such as missing a semicolon at the end of a statement or due to missing braces, class not found etc. These errors will be detected by java compiler & displays the errors on to the screen while compiling

Ex:

```
class err  
{  
    public static void main(String args[])  
    {  
        System.out.println("Hello").  
    }  
}
```

o/p : C:\javac err.java
Err: Java: 6, ';' expected

① Runtime errors: These errors represent inefficiency of the Computer System to execute a particular statement. For example, insufficient memory to store something or inability of microprocessor to execute some statement come under run-time errors. Runtime errors are not detected by the java compiler. They are detected by JVM, only at runtime.

/* program to write main() method without its parameter String args[].
Hence JVM can't detect & can't execute the code */

// Runtime error

class err

{ public static void main (~~String~~)

{ System.out.println ("Hello");

} }

o/p: C:\> javac Err.java

C:\> java Err

Exception in thread "main" java.lang.NoSuchMethodError: main

② Logical errors: These errors depicts flaws in the logic of the program. The programmer might be using a wrong formula or design of program itself is wrong. These ^{errors} are not detected by JVM or java compiler. The programmer is ~~sole~~ responsible for them.

Ex: class err

{ public static void main (String args[])

{ double sal = 5000.00;

sal = sal * 15/100; // wrong. use : salt = sal * 15/100.

System.out.println ("Incremented salary = " + sal);

} }

o/p

C:\> javac err.java

C:\> java err

Incremented salary = 750

• Exceptions :

An exception is a condition that is caused by a run-time error in the program. When the Java interpreter encounters an error such as dividing an integer by zero, it creates an exception object & throws it (i.e. inform us that an error has occurred).

If the exception object is not caught & handled properly, the interpreter will display an error message & will terminate the program as shown in example below:

```
// program without exception handling.
class Err
{
    public static void main (String args [])
    {
        int a=10;
        int b=5;
        int c=5;
        int x = a/(b-c); // division by zero
        System.out.println("x=" + x);
        int y = a/(b+c);
        System.out.println("y=" + y);
    }
}
```

The above program syntactically correct & ∴ doesn't cause any problem during compilation. However while executing it will display following message & stops without executing further statements.

C:\> javac err.java

C:\> java err

java.lang.ArithmeticException: / by zero
at Error2.main(Error2.java:10)

→ If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition & then display an appropriate message for taking corrective actions. This task is known as exception handling.

Java exception handling is managed via five keywords: try, catch, throw, throws & finally.

→ try ÷ The 'try' keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally.

→ catch ÷ The 'catch' block is used to handle the exception. It must be preceded by try block. It can be followed by finally block later.

→ finally ÷ Any code that absolutely must be executed after a try block completes is put in finally block. It is executed whether an exception is handled or not.

→ throw ÷ The 'throw' keyword is used to throw an exception manually.

→ throws ÷ The 'throws' keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

General form of exception handling block: (syntax of try-catch)

```
try
{
    statement; // generates an exception
}
catch (ExceptionType e)
{
    statement; // processes the exception
}
```

→ Here ExceptionType is the type of exception that has occurred

// using try & catch for exception handling.

```
Class err1
{
    public static void main (String args[])
    {
        int a=10;
        int b=5;
        int c=5;
        int x,y;
        try
        {
            x = a / (b - c);    // Exception here
        }
        catch (ArithmeticException e)
        {
            System.out.println ("Division by zero");
        }
        y = a / (b + c);
        System.out.println (" y = " + y);
    }
}
```

01) javac err1.java

c:\> java err1

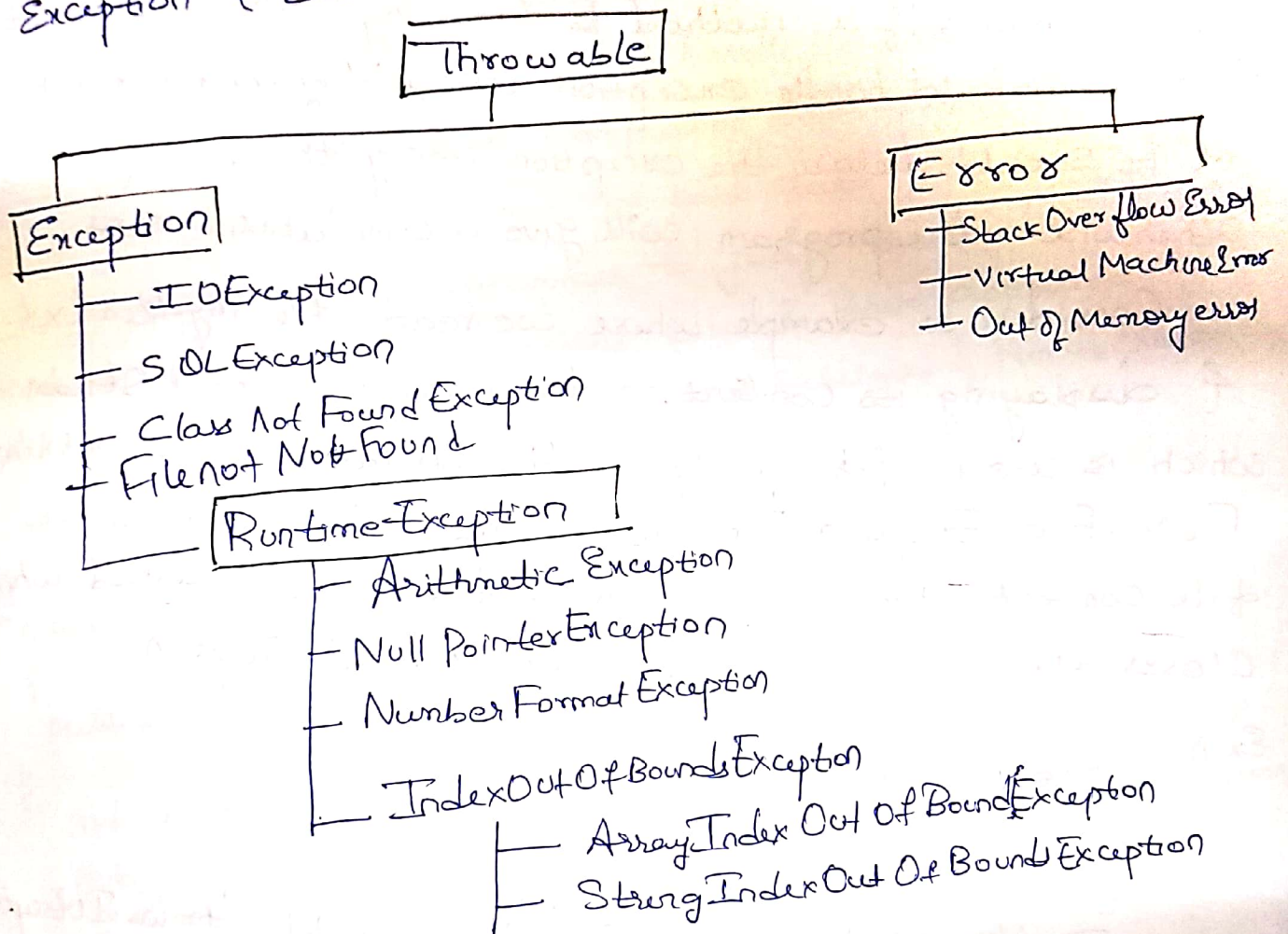
Division by zero

y=1

Note that the program did not stop at the point of exceptional condition. It catches the error condition, prints the error message, & then continues the execution.

Hierarchy of Java Exception classes

The java.lang Throwable class is the root class of java Exception hierarchy which is inherited by two subclasses.
Exception & Error



⇒ Types of Exceptions :

There are three types of exceptions

① Checked Exception ② Unchecked Exception ③ Error.

① Checked Exceptions : The classes which directly inherit Throwable class except RuntimeException & errors are known as Checked Exceptions eg. IOException, SQLException etc.
Checked exceptions are checked at compile time.

- A Checked exception is an exception that is checked by the compiler at compilation time, these are also called compile time exceptions. These exceptions can't simply be ignored, the programmer should take care of (handle) these exceptions.

It means if a method is throwing a checked exception then it should handle exception using try-catch block or it should declare the exception using throws keyword, otherwise the program will give a compilation error.

Consider an example where we read file myFile.txt & displaying its content on the screen. FileInputStream which is used for specifying the file path & name, throws FileNotFoundException. The read() method which reads the file content throws IOException & the close() method which closes the file input stream also throws IOException.

Ex:1)

```
import java.io.*;

class Example
{
    public static void main (String args []) throws IOException
    {
        FileInputStream fis = null;
        fis = new FileInputStream ("B:/myfile.txt");
        int k;
        while ((k = fis.read()) != -1)
        {
            System.out.println (char) k;
        }
        fis.close();
    }
}
```

→ IOException is a parent class of FileNotFoundException

① Unchecked Exception ÷ The classes which inherit RuntimeException are Unchecked Exceptions. Ex: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException. Unchecked Exceptions are not checked at compile time, they are checked by JVM at run-time.

It means if your program is throwing an unchecked exception, & even if you didn't handle/declare that exception, the program won't give compilation error. Most of the times these exceptions occur due to bad data provided by users. It is up to the programmer to judge the conditions in advance that can cause such exceptions & handle them appropriately. All unchecked exceptions are direct subclasses of RuntimeException.

Unchecked Exception Example

Ex 1:
Class Example

```
{  
    public static void main (String args[])
```

```
{  
    int num1 = 10;
```

```
    int num2 = 0;
```

/* Since we are dividing integer with 0 it should throw ArithmeticException */

```
    int res = num1 / num2;
```

```
    System.out.println(res);
```

```
}  
}
```

~~If~~ If you compile this code, it would compile successfully however when you will run it, it would throw ArithmeticException.

Ex 2

Class Example

```
{  
    public static void main (String args[])
```

```
{  
    int arr[] = {1, 2, 3, 4, 5};
```

/* array has only 5 elements but we are trying to display 8th element. It should throw ArrayIndexOutOfBoundsException */

```
    System.out.println (arr[7]);  
}  
}
```


Note: It doesn't mean that compiler is not checking these exceptions so we should not handle them. In fact we should handle them more carefully. For eg. In the above example there should be exception message to user that they are trying to display a value which doesn't exist in array so that user could be able to correct the issue.

```
class Example
```

```
{  
    public static void main (String args[])
```

```
{  
    try
```

```
{  
        int arr[] = {1, 2, 3, 4, 5};
```

```
        System.out.println (arr[7]);
```

```
    }
```

```
    catch (ArrayIndexOutOfBoundsException e)
```

```
{  
        System.out.println ("The specified index does not  
        exist" + " in array, please correct the error");
```

```
    }  
}
```

o/p: The specified index does not exist in array, please correct the error
(3) Error: error is irrecoverable. ex: out of memory, virtual machine error or stack overflow.

→ Default Exception handler:

```
class defaultexceptiondemo
```

```
{
```

```
    public static void main (String args[])
```

```
{
```

```
        abc();
```

```
}
```



```
public static void abc()
```

```
{ System.out.println(10/0);  
  xyz();
```

```
}
```

```
public static void abc()
```

```
{ System.out.println("hello");
```

```
}
```

O/p:
C:\> javac defaultExceptionDemo.java
C:\> java defaultExceptionDemo
Exception in thread main
java.lang.ArithmeticException: / by zero
at defaultException.abc(defaultEx.java:6)
at defaultException.main(defaultEx.java:3)

Inside a method if any exception occurs the method in which it is raised is responsible to create exception object by including following information
① name of exception ② Description ③ location at which exception occurs [Stack trace]

After creating exception object, method hands over that object to JVM. JVM will check whether the method contains any exception handling code or not. If the method does not contain exception handling code then JVM terminates that method abnormally & remove the corresponding entry from stack.

The JVM identifies caller method & checks whether caller method contains any handling code or not. If the caller method contains handling code then JVM terminates abnormally & removes corresponding entry from the stack. This process will be continued until main method & if the main method also abnormally doesn't contain handling code then JVM terminates main method abnormally & removes corresponding entry from the stack.

The JVM hands over responsibility of exception handling to default exception handler which is part of JVM.

→ Default Exceptional handler prints exception in the following format & terminates programs abnormally.

[Exception in thread 'xxx' Name of Exception Description Stack Trace]

* program illustrating exception handling to print custom message

class default test

```
{ public static void main (String args[])
```

```
{  
    abcc();
```

```
}
```

```
public static void abcc()
```

```
{
```

```
try
```

```
{ System.out.println(10/0);
```

```
}
```

```
catch (ArithmeticException e)
```

```
{ System.out.println("division by zero");
```

```
}
```

```
xyz();
```

```
}  
public static void xyz()
```

```
{ System.out.println("hello");
```

```
}
```

o/p : javac test.java
java test

division by zero
hello

Common scenarios of Java Exception:

There are some scenarios where unchecked exceptions may occur. They are as follows:

① A scenario where ArithmeticException occurs:
If we divide any number by zero, there occurs an Arithmetic Exception.

```
int a = 50/0; // Arithmetic Exception.
```

② Scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s = null;
```

```
System.out.println(s.length()); // NullPointerException
```

③ NumberFormatException

The wrong format of any value may occur NumberFormatException.
Ex. String variable in characters of converting into digits.
String s = "abc";

```
int i = Integer.parseInt(s); // NumberFormatException
```

④ ArrayIndexOutOfBoundsException

If we are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException.

```
Ex: int a[] = new int[5];  
a[10] = 50; // ArrayIndexOutOfBoundsException
```


⇒ Multiple-Catch block :

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if we have to perform different tasks at the occurrence of different exceptions, we use java multiple catch block.

Note :

- i) At a time only one exception occurs and at a time only one catch block is executed.
- ii) If try with multiple catch blocks present then the order of catch block is very important. we have to take child first & then parent (i.e. specific to most general) otherwise we get compile-time error saying
Exception xxx has already been caught.

/ * Example of Multi-catch block */

```
Public class Multicatch1
{
    Public static void main (String args[])
    {
        try
        {
            int a[] = new int[5];
            a[5] = 30/0;
        }
        catch (ArithmeticException e)
        {
            System.out.println ("Arithmetic Exception occurs");
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println ("Array Index out of Bounds error occurs");
        }
    }
}
```

```
catch (Exception e)
```

```
{ System.out.println("Parent Exception occurs");
```

```
} System.out.println("rest of code");
```

```
}
```

o/p: java multibatch
ArithmeticException occurs
rest of code

⇒/x In this example try block contains two exceptions. But at a time only one exception occurs & its corresponding catch block is invoked x/

```
class multibatch2
```

```
{ public static void main (String args[])
```

```
{ try
```

```
{ int a[] = new int[5];
```

```
  a[5] = 30/0;  
  System.out.println(a[10]);
```

```
} catch (ArithmeticException e)
```

```
{ System.out.println("Arithmetic exception occurs");
```

```
} catch (ArrayIndexOutOfBoundsException e)
```

```
{ System.out.println("ArrayIndex out of Bounds exception occurs");
```

```
} catch (Exception e)
```

```
{ System.out.println("Parent Exception occurs");
```

```
} System.out.println("rest of code");
```

```
}
```

o/p: ArithmeticException occurs
rest of code.

Ex: ~~note~~ / x In this example, we generate NullPointerException but did not provide corresponding exception type, in such case, the catch block containing parent exception class will be invoked.

```
class MultipleCatch4
{
    public static void main (String args[])
    {
        String s = null;
        System.out.println(s.length());
    }
    catch (ArithmeticException e)
    {
        System.out.println("error occurred");
    }
    catch (ArrayIndexOutOfBoundsException e)
    {
        System.out.println("error");
    }
    catch (Exception e)
    {
        System.out.println("parentException occurs");
    }
    System.out.println("rest of code");
}
```

Example to handle exception without maintaining the order of exception (i.e. from most specific to most generic)

```
class Multicatch4
{
    public static void main (String args[])
    {
        try
        {
            int a[5] = new int[5];
            a[5] = 30/0;
        }
        catch (Exception e)
        {
            System.out.println("Common task");
        }
        catch (ArithmeticException e)
        {
            System.out.println(e);
        }
    }
}
```

opp: Compile time error
exception ArithmeticException has
already been caught

Nested try-block:

try statement can be nested inside another block of try. Nested try block is used when a part of a block may cause one error while entire block may cause another error. In case if inner try block does not have a catch handler for a particular exception then the outer try catch block is checked for match.

/ * Program demonstrating Nested try statement */
class Ex.

```
{ public static void main (String args[])
```

```
{ try  
  { int a[] = {5, 0, 1, 2};
```

```
    try  
    { int x = a[3]/a[1];
```

```
    }  
  catch (ArithmeticException e)
```

```
  { System.out.println("divide by zero");
```

```
  }  
  a[4] = 3;
```

```
  }  
  catch (ArrayIndexOutOfBoundsException e)
```

```
  { System.out.println("Array index out of bounds exception");
```

```
  }  
}
```

O/p: divide by zero
Array index out bounds exception.

⇒ finally statement

- finally block is used to place ^{execute} important code such as closing connection, stream etc.
- finally block is always executed whether exception is handled or not.
- finally block is always associated with try or catch block.

Syntax ÷

```
try
{
    statement; // generates exception
}
catch (Exception type e)
{
    statement
}
finally
{
    statement
}
```

Note: If we don't handle exception, before terminating the program, JVM executes finally block (if any)

Case 1: Different cases where finally can be used
program demonstrating finally where exception occurs & not handled

```
class finallydemo
{
    public static void main (String args[])
    {
        try
        {
            int data = 25/0;
            System.out.println (data);
        }
        catch (NullPointerException e)
        {
            System.out.println (e);
        }
        finally
        {
            System.out.println ("finally block is always executed");
        }
        System.out.println ("rest of code");
    }
}
```

O/p: finally block is always executed

ArithmeticException in thread main java.lang.ArithmeticException: / by zero

/* example where exception doesn't occur */

ex2: class finallydemo2

```
{ public static void main (String args[])
```

```
{ try
```

```
{ int data = 25/5;
```

```
System.out.println(data);
```

```
} catch (NullPointerArithmeticException e)
```

```
{ System.out.println(e);
```

```
}
```

```
finally
```

```
{ System.out.println("finally block is always executed");
```

```
}
```

```
System.out.println("rest of code");
```

```
}
```

O/p: 5 finally block is always executed
rest of code

case 3: // example where exception occurs & handled

class finallyblock3

```
{ public static void main (String args[])
```

```
{ try
```

```
{ int data = 25/0;
```

```
System.out.println(data);
```

```
} catch (ArithmeticException e)
```

```
{ System.out.println(e);
```

```
}
```

```
finally
```

```
{ System.out.println("finally always executed");
```

```
} System.out.println("rest of code");
```

```
}
```

O/p: java.lang.ArithmeticException: / by zero
finally block is always executed
rest of code.

Note: for each try block there can be zero or more catch block, but only one finally block.

Lab 10.1

* program to demonstrate exception handling by using
Throw, finally & multiple catch statements */

class multiplecatchdemo

```
{  
    public static void main(String args[])
```

```
{  
    try
```

```
{  
        int a[] = new int[5];
```

```
        a[5] = 9;
```

```
    }  
    catch (ArithmeticException e)
```

```
{  
        System.out.println("arithmetic exception occurs");
```

```
    }  
    catch (ArrayIndexOutOfBoundsException e)
```

```
{  
        System.out.println("ArrayIndexOutOfBoundsException occurs");
```

```
    }  
    catch (Exception e)
```

```
{  
        System.out.println("exception occurs");
```

```
    }
```

```
    finally
```

```
{  
        System.out.println("finally block executed");
```

```
    }  
    System.out.println("outside try-catch-finally clause");
```

```
}  
}
```

O/p: java multiplecatchdemo

ArrayIndexOutOfBoundsException occurs:

finally block executed

outside try-catch-finally clause.

26/11/19

→ Throwing our own Exceptions (Userdefined Exceptions)

• throw :

- throw keyword in java is used to explicitly throw an exception from a method or any block of code.
- we can throw either checked or unchecked exception.
- The throw keyword is mainly used to throw custom (userdefined) exceptions.

General form of throw :

throw ThrowableInstance

- Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

There are two ways you can obtain Throwable object :
using a parameter in a catch clause or creating one with the new operator.

Ex : throw new ArithmeticException("/ by zero");

But this exception i.e, instance method must be of type Throwable or a subclass of Throwable. For ex, exception is a subclass of Throwable & userdefined exceptions typically extend Exception class.

The flow of execution of program stops immediately after the throw statement is executed & the nearest enclosing try block is checked to see if it has a catch statement that matches the types of exception. If it finds a match, control is transferred to that statement otherwise next enclosing try block is checked & soon. if no matching catch is found the default exception handler will halt the program.

Ex¹ /* Program demonstrating throw keyword, in this example we have created validate method that takes integer value as parameter. If age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote */

```
class testthrow
{
    static void validate(int age)
    {
        if (age < 18)
        {
            throw new ArithmeticException("Not valid to give vote");
        }
        else
        {
            System.out.println("welcome to vote");
        }
    }
    public static void main(String args[])
    {
        validate(Integer.parseInt(args[0]));
        System.out.println("testing complete");
    }
}
```

o/p: }

C:\> java c testthrow.java

C:\> java testthrow 12

Exception in thread main java.lang.ArithmeticException: not valid to give vote

C:\> java testthrow 19

welcome to vote

testing complete

/* handle the exception */
class testthrow

```
{
    static void validate(int age)
    {
        if (age < 18)
        {
            throw new ArithmeticException("Not valid to give vote");
        }
        else
        {
            System.out.println("welcome to vote");
        }
    }
}
```



```
public static void main (String args[])
```

```
{ try
```

```
{ validate ( Integer.parseInt ( args[0] );
```

```
{ catch ( ArithmeticException e)
```

```
{ System.out.println (e)
```

```
{ System.out.println ( "Division by zero" );
```

```
{ System.out.println ( "Testing complete" );
```

```
}  
} // Demonstrate throw  
class throwdemo
```

```
{ static void demoproc()
```

```
{ try
```

```
{ throw new NullPointerException ("demo");
```

```
{ catch (NullPointerException e)
```

```
{ System.out.println ( "Caught inside demoproc" );
```

```
throw e; // Rethrow the exception
```

```
}  
}
```

```
public static void main (String args[])
```

```
{ try
```

```
{ demoproc();
```

```
{ catch (NullPointerException e)
```

```
{ System.out.println ( "Recought : " + e );
```

```
}  
}
```

O/p: java testthrow 12
= java.lang.ArithmeticException: Not
division by zero
Valid to give vote
-testing complete

O/p:
Caught: Inside demoproc
Recought: java.lang.NullPointerException
Exception demo.

This program gets two chances to deal with same error. First main() sets up an exception context & then calls demoproc(). The demoproc() method then sets up another exception handling context & immediately throws a new instance of NullPointerException, which is caught on the next line. The exception is then rethrown.

- throw new NullPointerException("demo");

→ Here, the new is used to construct an instance of NullPointerException.

→ The argument specifies a string that describes the exception.

⇒ throws keyword ÷ it is used to declare exception

throws is a keyword in java which is used in signature of method to indicate that this method might throw one of the listed type of exceptions. The caller to these methods has to handle the exception using a try-catch block.

Syntax ÷

```
type method-name(parameters) throws exceptionlist  
{  
    // body of method  
}
```

exception list is a comma separated list of all the exceptions which a method might throw

— A throws clause lists the types of exceptions that a method might throw. This is necessary for all the exceptions, except those of type Error or RuntimeException, or any of their subclasses (that means necessary for checked exceptions).

— In a program, if there is a chance of raising an exception then compiler always warns^u about it & compulsorily we should handle that checked exception, otherwise we will get compile time error saying unreported exception xxx must be caught or declared to be thrown.

To prevent this compile time error we can handle the exception in two ways:

① By using try catch

② By using throws keyword.

Q1/ * Program demonstrating throws keyword. In this example the method myMethod() is throwing two checked exceptions. So we have declared these exceptions in the method signature using throws keyword. If we don't declare these exceptions then program will throw a compilation error.

```
import java.io.*;

class ThrowExample
{
    void myMethod(int num) throws IOException, ClassNotFoundException
    {
        if (num == 1)
        {
            throw new IOException("IOException occurred");
        }
        else
        {
            throw new ClassNotFoundException("ClassNotFoundException occurred");
        }
    }
}

public class example
{
    public static void main (String args[])
    {
        try
        {
            ThrowExample ob = new ThrowExample();
            ob.myMethod(1);
        }
        catch (Exception ex)
        {
            System.out.println(ex);
        }
    }
}
```

O/p :

```
java < example.java
java example
java.io.IOException: IOException occurred.
```


Ex 2:

/ * program to demonstrate working of throws */
class throws example

```
{  
    static void func() throws IllegalAccess Exception  
    {  
        System.out.println("Inside func()");  
        throw new IllegalAccess Exception("demo");  
    }  
    public static void main (String args [])  
    {  
        try  
        {  
            func();  
        }  
        catch (IllegalAccess Exception e)  
        {  
            System.out.println("Caught in main");  
        }  
    }  
}
```

O/p:
Inside func()
Caught in main

Ex 3:

class myexception extends Exception

```
{  
    myexception (String s)
```

```
{  
    super(s);  
}
```

```
}
```

class excep

```
{  
    static void validate (int age) throws myexception
```

```
{  
    if (age < 18)
```

```
{  
        throw new myexception("not valid to give vote");  
    }
```

```
}
```

```

else
{
    System.out.println("welcome to vote");
}
}
public static void main(String args[])
{
    try
    {
        validate(Integer.parseInt(args[0]));
    }
    catch (MyException my)
    {
        System.out.println(my);
    }
    System.out.println("testing complete");
}
}
}

```

12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

java c excep.java
 java excep 12
 myexception: not valid to give vote
 testing complete
 java excep 19
 welcome to vote
 testing complete
 → without using throws keyword we can't propagate checked
 exception.

Ex 2: // Creating our own Exception.
class wrongwordexception extends Exception

```
{  
    String myword;  
    wrongwordexception (String a) // Constructor  
    {  
        myword = a;  
    }  
    String myprint()   
    {  
        return "Exception caught because word is " + myword;  
    }  
}
```

```
class myownexcep  
{  
    public static void main (String args[])  
    {  
        try  
        {  
            userword ("peace");  
            userword ("war");  
        }  
        catch (wrongwordexception e)  
        {  
            System.out.println ("wrongword:" + e.myprint());  
        }  
    }  
    static void userword (String myword) throws wrongwordexception  
    {  
        if (myword.equals ("war"))  
        {  
            throw new wrongwordexception (myword);  
        }  
        else  
        {  
            System.out.println ("Correct word");  
        }  
    }  
}
```


o/p ÷ javac myownexcep.java

java myownexcep

correct word

wrong word: Exception caught because word is war.

→ Difference between throw & throws

- | | |
|---|---|
| <u>throw</u> | <u>throws</u> |
| (i) java <u>throw</u> keyword is used to explicitly <u>throw</u> an exception | (i) java <u>throws</u> keyword is used to declare an exception |
| (ii) checked exception can't be propagated using <u>throw</u> only | (ii) checked exception can be propagated with <u>throws</u> |
| (iii) <u>throw</u> is followed by an instance | (iii) <u>throws</u> is followed by class |
| (iv) <u>throw</u> is used with method | (iv) <u>throws</u> is used ^{method} _{signature} |
| (v) multiple you can't <u>throw</u> multiple exceptions | (v) you can declare multiple exceptions. |
- Ex: public void method() throws IOException, SQLException.

→ Garbage Collection:

- Garbage collection means unreferenced objects.
- It is a process of reclaiming the runtime unused memory automatically. In other words it is a way to destroy unused object.
- JVM decides when to run Garbage collector. Garbage collector is under control of JVM. From the java program we can tell JVM to run Garbage collector.

Advantages :

- ① It makes memory efficient because garbage collector removes unreferenced object from the heap memory.
- ② It is automatically done by garbage collector so we don't need to make extra efforts.

Following ways to make object eligible for garbage collection

- ① By nullifying reference.

Ex: `emp e = new emp();`
`e = null;`

- ② By assigning a reference to another.

`emp e1 = new emp();`
`emp e2 = new emp();`
`e1 = e2();` // now first object referenced by e1 is available for garbage collector.

⇒ finalize() method :

`finalize()` method is invoked by garbage collector just before destroying an object to perform clean up activities. Once `finalize()` method completes immediately Garbage collector destroy object.

`gc()` : This method is used to invoke garbage collector. It is in ~~System~~^{System} & Runtime class.

```
// Program demonstrating finalize()

public class GarbageC
{
    void f1()
    {
        System.out.println("f1 method");
    }
    void finalize()
    {
        System.out.println("Object is garbage
        collected");
    }
}

public static void main(String args[])
{
    GarbageC s1 = new GarbageC();
    GarbageC s2 = new GarbageC();
    s1 = null;
    s2.f1();
    System.gc();
}
}
```

O/p: f1 method
Object is garbage collected.

Note:

- Finally block is responsible to perform cleanup activities related to try block i.e. whatever resources are opened as part of try block will be checked inside finally block.
- Where as `finalize()` method is responsible to perform cleanup activities related to object i.e. whatever resources associated with object will be deallocated before destroying an object by using `finalize()` method.

Difference between final, finally, finalize()

final

- ① final is used to apply restrictions on class, method & variable. final class can't be inherited, final method can't be overridden & final variable value can't be changed.

- ② final is keyword

finally

- ① finally is used to place important code, it will be executed whether exception is handled or not.

- ② finally is block.

finalize()

- ① `finalize()` is used to perform cleanup & processing before object is garbage collected.

- ② `finalize` is a method.

Exception Handling in Java

The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

In this page, we will learn about java exception, its type and the difference between checked and unchecked exceptions.

What is exception

Dictionary Meaning: Exception is an abnormal condition.

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is exception handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IO, SQL, Remote etc.

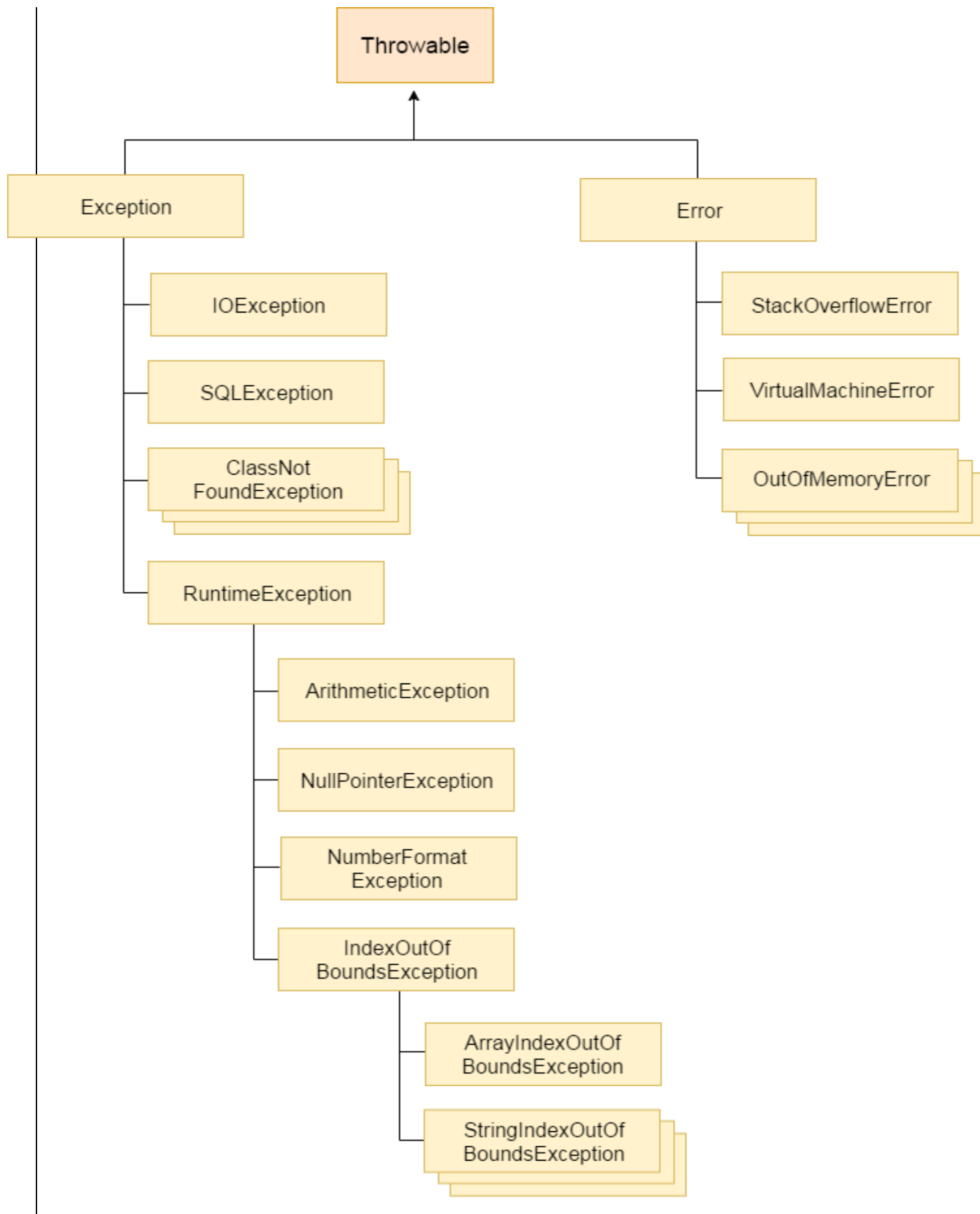
Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

1. statement 1;
2. statement 2;
3. statement 3;
4. statement 4;
5. statement 5; *//exception occurs*
6. statement 6;
7. statement 7;
8. statement 8;
9. statement 9;
10. statement 10;

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

Hierarchy of Java Exception classes



Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between checked and unchecked exceptions

1) Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Common scenarios where exceptions may occur

There are given some scenarios where unchecked exceptions can occur. They are as follows:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

1. `int a=50/0;//ArithmeticException`
-

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

1. String s=**null**;
 2. System.out.println(s.length());**//NullPointerException**
-

3) Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

1. String s="**abc**";
 2. **int** i=Integer.parseInt(s);**//NumberFormatException**
-

4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

1. **int** a[]=**new int**[**5**];
 2. a[**10**]=**50**; **//ArrayIndexOutOfBoundsException**
-

Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws

Java try-catch

Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

Java try block must be followed by either catch or finally block.

Syntax of java try-catch

1. **try**{
2. *//code that may throw exception*
3. }**catch**(Exception_class_Name ref){}

Syntax of try-finally block

1. **try**{
2. *//code that may throw exception*
3. }**finally**{}

Java catch block

Java catch block is used to handle the Exception. It must be used after the try block only.

You can use multiple catch block with a single try.

Problem without exception handling

Let's try to understand the problem if we don't use try-catch block.

1. **public class** Testtrycatch1{
2. **public static void** main(String args[]){
3. **int** data=50/0;*//may throw exception*
4. System.out.println("rest of the code...");
5. }
6. }

Test it Now

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
```


As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

Let's see the solution of above problem by java try-catch block.

1. **public class** Testtrycatch2{
2. **public static void** main(String args[]){
3. **try**{
4. **int** data=50/0;
5. }**catch**(ArithmeticException e){System.out.println(e);}
6. System.out.println("rest of the code...");
7. }
8. }

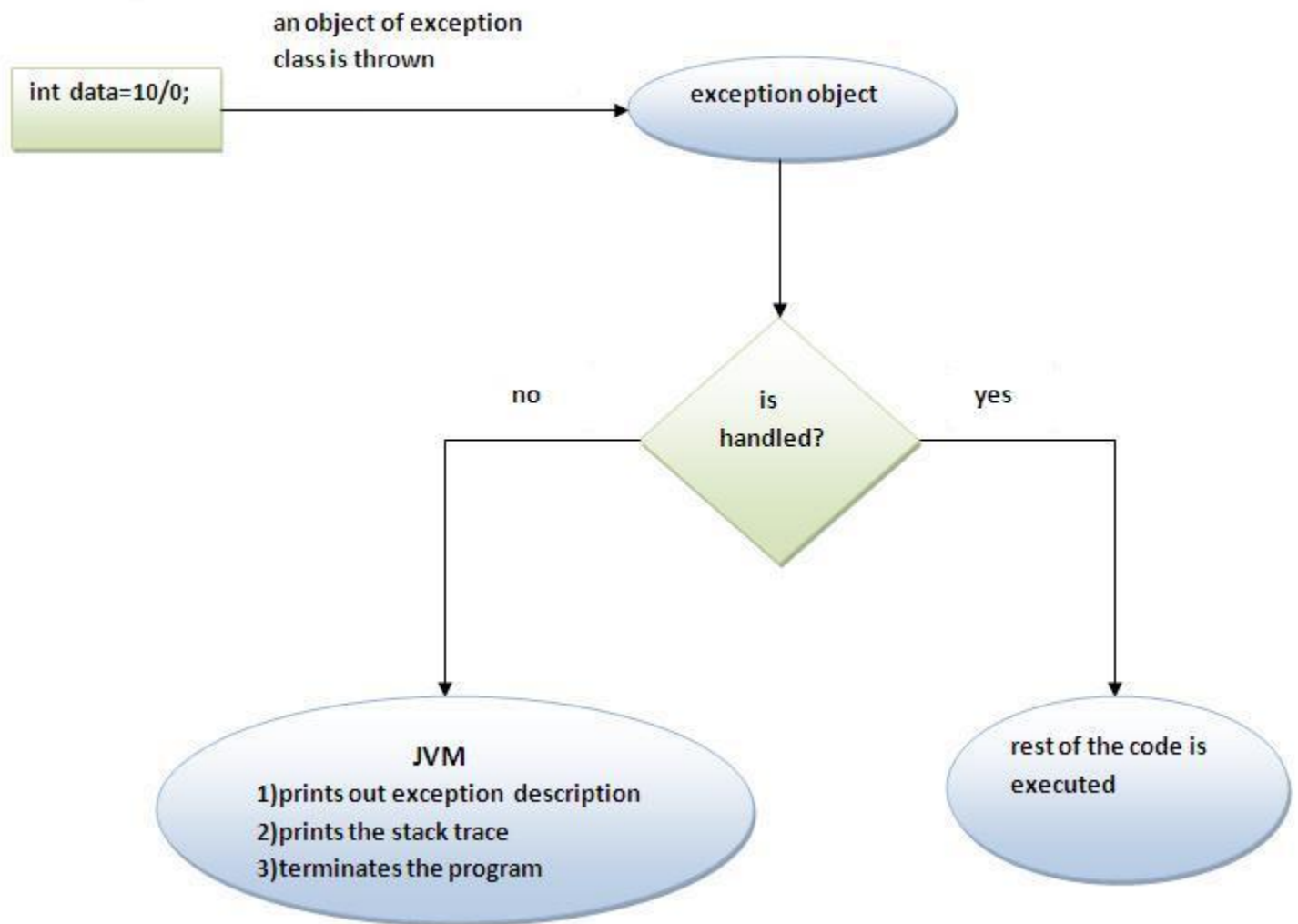
[Test it Now](#)

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

Internal working of java try-catch block



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

Java catch multiple exceptions

Java Multi catch block

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

Let's see a simple example of java multi-catch block.

```
1. public class TestMultipleCatchBlock{
2.     public static void main(String args[]){
3.         try{
4.             int a[]=new int[5];
5.             a[5]=30/0;
6.         }
7.         catch(ArithmeticException e){System.out.println("task1 is completed");}
8.         catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
9.         catch(Exception e){System.out.println("common task completed");}
10.
11.     System.out.println("rest of the code...");
12. }
13. }
```

Test it Now

```
Output:task1 completed
        rest of the code...
```

Rule: At a time only one Exception is occurred and at a time only one catch block is executed.

Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .

```
1. class TestMultipleCatchBlock1{
2.     public static void main(String args[]){
3.         try{
4.             int a[]=new int[5];
5.             a[5]=30/0;
6.         }
7.         catch(Exception e){System.out.println("common task completed");}
8.         catch(ArithmeticException e){System.out.println("task1 is completed");}
9.         catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
10.     System.out.println("rest of the code...");
}
```


11. }

12. }

Test it Now

Output:

Compile-time error

Java Nested try block

The try block within a try block is known as nested try block in java.

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
1. ....
2. try
3. {
4.     statement 1;
5.     statement 2;
6.     try
7.     {
8.         statement 1;
9.         statement 2;
10.    }
11.    catch(Exception e)
12.    {
13.    }
14. }
15. catch(Exception e)
16. {
17. }
18. ....
```

Java nested try example

Let's see a simple example of java nested try block.

```
1. class Excep6{
2.     public static void main(String args[]){
3.         try{
4.             try{
5.                 System.out.println("going to divide");
6.                 int b = 39/0;
7.             }catch(ArithmeticException e){System.out.println(e);}
8.
9.             try{
10.                int a[]=new int[5];
11.                a[5]=4;
12.            }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}
13.
14.            System.out.println("other statement");
15.        }catch(Exception e){System.out.println("handeled");}
16.
17.        System.out.println("normal flow..");
18.    }
19. }
```

Java is an object-oriented programming language. It allows you to divide complex problems into smaller sets by creating objects.

These objects share two characteristics:

- state
- behavior

Let's take few examples:

1. Lamp is an object
 - It can be in on or off state.
 - You can turn on and turn off lamp (behavior).
2. Bicycle is an object
 - It has current gear, two wheels, number of gear etc. states.
 - It has braking, accelerating, changing gears etc. behavior.

You will learn about 3 main features of an object-oriented programming: *data encapsulation*, *inheritance* and *polymorphism* in later chapters. This article will focus on class and objects to keep things simple.

Recommended reading: [What is an object?](#)

Java Class

Before you create objects in Java, you need to define a class.

A class is a blueprint for the object.

We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object. Since, many houses can be made from the same description, we can create many objects from a class.

How to define a class in Java?

Here's how a class is defined in Java:

```
class ClassName {  
    // variables  
    // methods  
}
```

Here's an example:

```
class Lamp {  
  
    // instance variable  
    private boolean isOn;  
  
    // method  
    public void turnOn() {  
        isOn = true;  
    }  
  
    // method  
    public void turnOff() {
```



```
        isOn = false;
    }
}
```

Here, we defined a class named `Lamp`.

The class has one instance variable (variable defined inside class) `isOn` and two methods `turnOn()` and `turnOff()`. These variables and methods defined within a class are called **members** of the class.

Notice two keywords, `private` and `public` in the above program. These are access modifiers which will be discussed in detail in later chapters. For now, just remember:

- The `private` keyword makes instance variables and methods private which can be accessed only from inside the same class.
- The `public` keyword makes instance variables and methods public which can be accessed from outside of the class.

In the above program, `isOn` variable is private whereas `turnOn()` and `turnOff()` methods are public.

If you try to access `private` members from outside of the class, compiler throws error.

Java Objects

When class is defined, only the specification for the object is defined; no memory or storage is allocated.

To access members defined within the class, you need to create objects. Let's create objects of `Lamp` class.

```
class Lamp {
    boolean isOn;

    void turnOn() {
        isOn = true;
    }

    void turnOff() {
        isOn = false;
    }
}

class ClassObjectsExample {
    public static void main(String[] args) {
        Lamp l1 = new Lamp(); // create l1 object of Lamp class
        Lamp l2 = new Lamp(); // create l2 object of Lamp class
    }
}
```

This program creates two objects `l1` and `l2` of class `Lamp`.

How to access members?

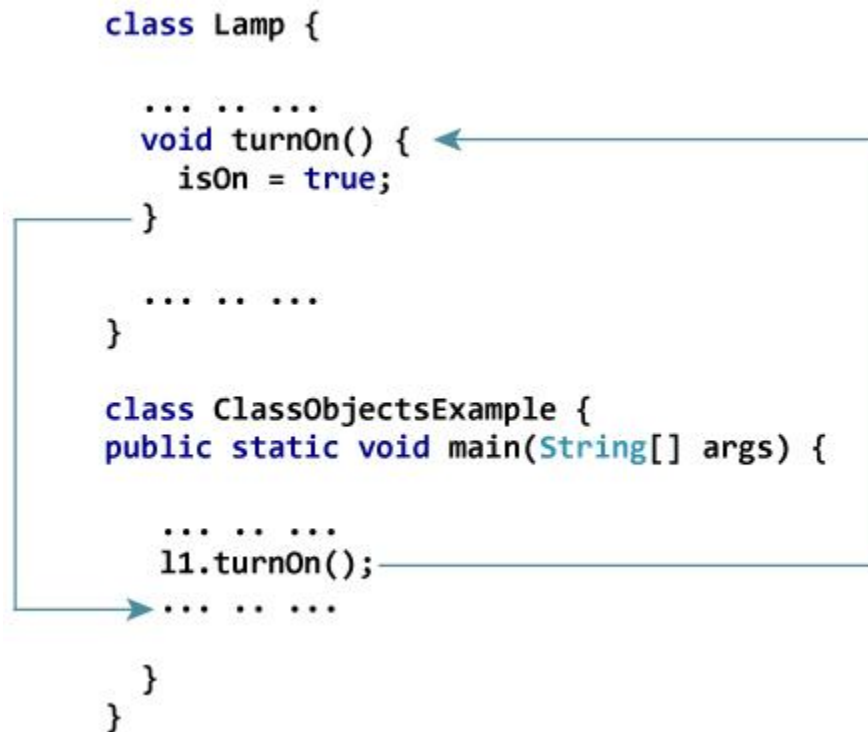
You can access members (call methods and access instance variables) by using `.` operator. For example,

```
l1.turnOn();
```

This statement calls `turnOn()` method inside `Lamp` class for `l1` object.

We have mentioned word **method** quite a few times. You will learn about *Java methods* in detail in the next chapter. Here's what you need to know for now:

When you call the method using the above statement, all statements within the body of `turnOn()` method are executed. Then, the control of program jumps back to the statement following `l1.turnOn();`



Similarly, the instance variable can be accessed as:

```
l2.isOn = false;
```

It is important to note that, the `private` members can be accessed only from inside the class. If the code `l2.isOn = false;` lies within the `main()` method (outside of the `Lamp` class), compiler will show error.

Example: Java Class and Objects

```
class Lamp {
    boolean isOn;

    void turnOn() {
        isOn = true;
    }

    void turnOff() {
        isOn = false;
    }

    void displayLightStatus() {
        System.out.println("Light on? " + isOn);
    }
}

class ClassObjectsExample {
    public static void main(String[] args) {

        Lamp l1 = new Lamp(), l2 = new Lamp();

        l1.turnOn();
        l2.turnOff();

        l1.displayLightStatus();
        l2.displayLightStatus();
    }
}
```

When you run the program, the output will be:

```
Light on? true
Light on? false
```

In the above program,

- Lamp class is created.
- The class has an instance variable `isOn` and three methods `turnOn()`, `turnOff()` and `displayLightStatus()`.
- Two objects `l1` and `l2` of `Lamp` class are created in the `main()` function.
- Here, `turnOn()` method is called using `l1` object: `l1.turnOn()`;
- This method sets `isOn` instance variable of `l1` object to `true`.
- And, `turnOff()` method is called using `l2` object: `l2.turnOff()`;
- This method sets `isOff` instance variable of `l2` object to `false`.

- Finally, `l1.displayLightStatus();` statement displays `Light on?` `true` because `isOn` variable holds `true` for `l1` object.
- And, `l2.displayLightStatus();` statement displays `Light on?` `false` because `isOn` variable holds `false` for `l2` object

Note, variables defined within a class are called **instance variable** for a reason.

When an object is initialized, it's called an instance. Each instance contains its own copy of these variables. For example, `isOn` variable for objects `l1` and `l2` are different.

What is a method?

In mathematics, you might have studied about functions. For example, $f(x) = x^2$ is a function that returns squared value of x .

If $x = 2$, then $f(2) = 4$

If $x = 3$, $f(3) = 9$

and so on.

Similarly, in programming, a function is a block of code that performs a specific task.

In object-oriented programming, method is a jargon used for function. Methods are bound to a class and they define the behavior of a class.

Recommended Reading: [Java Class and Objects](#)

Types of Java methods

Depending on whether a method is defined by the user, or available in standard library, there are two types of methods:

- Standard Library Methods
- User-defined Methods

Standard Library Methods

The standard library methods are built-in methods in Java that are readily available for use. These standard libraries come along with the Java Class Library (JCL) in a Java archive (*.jar) file with JVM and JRE.

For example,

- `print()` is a method of `java.io.PrintStream`. The `print("...")` prints the string inside quotation marks.
- `sqrt()` is a method of `Math` class. It returns square root of a number.

Here's an working example:

```
public class Numbers {  
    public static void main(String... args) {  
        System.out.print("Square root of 4 is: " + Math.sqrt(4));  
    }  
}
```

When you run the program, the output will be:

```
Square root of 4 is: 2.0
```

User-defined Method

You can also define methods inside a class as per your wish. Such methods are called user-defined methods.

How to create a user-defined method?

Before you can use (call a method), you need to define it.

Here is how you define methods in Java.

```
public static void myMethod() {  
  
    System.out.println("My Function called");  
  
}
```

Here, a method named `myMethod()` is defined.

You can see three keywords `public`, `static` and `void` before the function name.

- The `public` keyword makes `myMethod()` method `public`. Public members can be accessed from outside of the class. To learn more, visit: [Java public and private Modifiers](#).
- The `static` keyword denotes that the method can be accessed without creating the object of the class. To learn more, visit: *Static Keyword in Java*
- The `void` keyword signifies that the method doesn't return any value. You will learn about returning value from the method later in this article.

In the above program, our method doesn't accept any arguments. Hence the empty parenthesis (). You will learn about passing arguments to a method later in this article.

The complete syntax for defining a Java method is:

```
modifier returnType static nameOfMethod (Parameter List) {  
  
    // method body  
  
}
```

Here,

- **modifier** - defines access type whether the method is public, private and so on.
- **returnType** - A method can return a value.

It can return native data types (int, float, double etc.), native objects (String, Map, List etc.), or any other built-in and user defined objects.

If the method does not return a value, its return type is void.

- **static** - If you use `static` keyword in a method then it becomes a static method. Static methods can be called without creating an instance of a class.

For example, the `sqrt()` method of standard [Math class](#) is static. Hence, we can directly call `Math.sqrt()` without creating an instance of Math class.

- **nameOfMethod** - The name of the method is an [identifier](#).

You can give any name to a method. However, it is more conventional to name it after the tasks it performs. For example, `calculateInterest`, `calculateArea`, and so on.

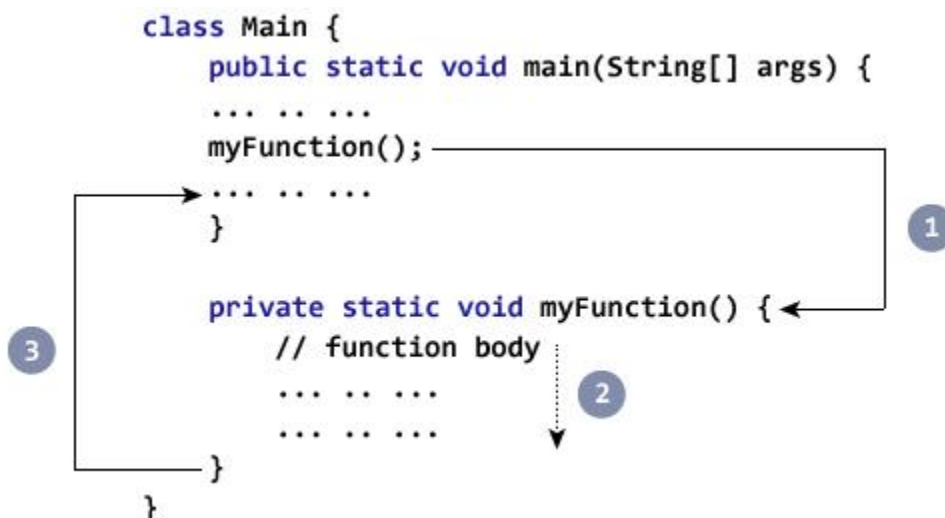
- **Parameters (arguments)** - Parameters are the values passed to a method. You can pass any number of arguments to a method.
- **Method body** - It defines what the method actually does, how the parameters are manipulated with programming statements and what values are returned. The codes inside curly braces { } is the body of the method.

How to call a Java Method?

Now you defined a method, you need to use it. For that, you have to call the method. Here's how:

```
myMethod();
```

This statement calls the `myMethod()` method that was declared earlier.



1. While Java is executing the program code, it encounters `myMethod();` in the code.
2. The execution then branches to the `myFunction()` method, and executes code inside the body of the method.
3. After the codes execution inside the method body is completed, the program returns to the original state and executes the next statement.

Example: Complete Program of Java Method

Let's see a Java method in action by defining a Java class.

```
class Main {  
  
    public static void main(String[] args) {  
        System.out.println("About to encounter a method.");  
  
        // method call  
        myMethod();  
  
        System.out.println("Method was executed successfully!");  
    }  
  
    // method definition  
    private static void myMethod(){  
        System.out.println("Printing from inside myMethod()!");  
    }  
}
```

When you run the program, the output will be:

About to encounter a method.

Printing from inside myMethod().

Method was executed successfully!

The method `myMethod()` in the above program doesn't accept any arguments. Also, the method doesn't return any value (return type is `void`).

Note that, we called the method without creating object of the class. It was possible because `myMethod()` is static.

Here's another example. In this example, our method is non-static and is inside another class.

```
class Main {

    public static void main(String[] args) {

        Output obj = new Output();
        System.out.println("About to encounter a method.");

        // calling myMethod() of Output class
        obj.myMethod();

        System.out.println("Method was executed successfully!");
    }
}

class Output {

    // public: this method can be called from outside the class
    public void myMethod() {
        System.out.println("Printing from inside myMethod().");
    }
}
```


When you run the program, the output will be:

```
About to encounter a method.
```

```
Printing from inside myMethod().
```

```
Method was executed successfully!
```

Note that, we first created instance of `Output` class, then the method was called using `obj` object. This is because `myMethod()` is a non-static method.

Java Methods with Arguments and Return Value

A Java method can have zero or more parameters. And, they may return a value.

Example: Return Value from Method

Let's take an example of method returning a value.

```
class SquareMain {  
    public static void main(String[] args) {  
        int result;  
        result = square();  
        System.out.println("Squared value of 10 is: " + result);  
    }  
  
    public static int square() {  
        // return statement  
        return 10 * 10;  
    }  
}
```

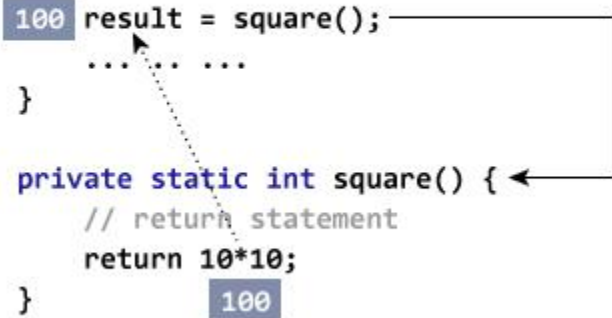
When you run the program, the output will be:

```
Squared value of 10 is: 100
```

In the above code snippet, the method `square()` does not accept any arguments and always returns the value of 10 squared.

Notice, the return type of `square()` method is `int`. Meaning, the method returns an integer value.

```
class SquareMain {  
    public static void main(String[] args) {  
        ... ..  
        100 result = square();  
        ... ..  
    }  
  
    private static int square() {  
        // return statement  
        return 10*10;  
    }  
}
```



As you can see, the scope of this method is limited as it always returns the same value.

Now, let's modify the above code snippet so that instead of always returning the squared value of 10, it returns the squared value of any integer passed to the method.

Example: Method Accepting Arguments and Returning Value

```
public class SquareMain {  
  
    public static void main(String[] args) {  
        int result, n;  
    }  
}
```

```
n = 3
result = square(n);
System.out.println("Square of 3 is: " + result);

n = 4
result = square(n);
System.out.println("Square of 4 is: " + result);
}

static int square(int i) {
    return i * i;
}
}
```

When you run the program, the output will be:

Squared value of 3 is: 9

Squared value of 4 is: 16

Now, the `square()` method returns the squared value of whatever integer value passed to it.


```

class SquareMain {
    public static void main(String[] args) {
        ... ..
        n = 3;
        9 result = square(n);
        ... ..
    }

    private static int square(int i) {
        // return statement
        return i*i;
    }
}

```

Java is a strongly-typed language. If you pass any other data type except `int` (in the above example), compiler will throw an error.

The argument passed `n` to the `getSquare()` method during the method call is called actual argument.

```
result = getSquare(n);
```

The parameter `i` accepts the passed arguments in the method definition `getSquare(int i)`. This is called formal argument (parameter). The type of the formal argument must be explicitly typed.

You can pass more than one argument to the Java method by using commas. For example,

```

public class ArithmeticMain {

    public static int getIntegerSum (int i, int j) {
        return i + j;
    }

    public static int multiplyInteger (int x, int y) {
        return x * y;
    }
}

```

```

public static void main(String[] args) {
    System.out.println("10 + 20 = " + getIntegerSum(10, 20));
    System.out.println("20 x 40 = " + multiplyInteger(20, 40));
}
}

```

When you run the program, the output will be:

```
10 + 20 = 30
```

```
20 x 40 = 800
```

The data type of actual and formal arguments should match, i.e., the data type of first actual argument should match the type of first formal argument. Similarly, the type of second actual argument must match the type of second formal argument and so on.

Example: Get Squared Value Of Numbers from 1 to 5

```

public class JMethods {

    // method defined
    private static int getSquare(int x){
        return x * x;
    }

    public static void main(String... args) {
        for (int i = 1; i <= 5; i++) {

            // method call
            result = getSquare(i)

```

```
        System.out.println("Square of " + i + " is : " + result); }  
    }  
}
```

When you run the program, the output will be:

Square of 1 is : 1

Square of 2 is : 4

Square of 3 is : 9

Square of 4 is : 16

Square of 5 is : 25

In above code snippet, the method `getSquare()` takes `int` as a parameter. Based on the argument passed, the method returns the squared value of it.

Here, argument `i` of type `int` is passed to the `getSquare()` method during method call.

```
result = getSquare(i);
```

The parameter `x` accepts the passed argument [in the function definition `getSquare(int x)`].

`return i * i;` is the return statement. The code returns a value to the calling method and terminates the function.

Did you notice, we reused the `getSquare` method 5 times?

What are the advantages of using methods?

- The main advantage is code reusability. You can write a method once, and use it multiple times. You do not have to rewrite the entire code each time. Think of it as, "write once, reuse multiple times."
- Methods make code more readable and easier to debug. For example, `getSalaryInformation()` method is so readable, that we can know what this method will be doing than actually reading the lines of code that make this method.

In Java, two or more [methods](#) can have same name if they differ in parameters (different number of parameters, different types of parameters, or both). These methods are called overloaded methods and this feature is called method overloading. For example:

```
void func() { ... }

void func(int a) { ... }

float func(double a) { ... }

float func(int a, float b) { ... }
```

Here, `func()` method is overloaded. These methods have same name but accept different arguments.

Notice that, the return type of these methods are not same. Overloaded methods may or may not have different return type, but they must differ in parameters they accept.

Why method overloading?

Suppose, you have to perform addition of the given numbers but there can be any number of arguments (let's say either 2 or 3 arguments for simplicity).

In order to accomplish the task, you can create two methods `sum2num(int, int)` and `sum3num(int, int, int)` for two and three parameters respectively. However, other programmers as well as you in future may get confused as the behavior of both methods is same but they differ by name.

The better way to accomplish this task is by overloading methods. And, depending upon the argument passed, one of the overloaded methods is called. This helps to increase readability of the program.

How to perform method overloading in Java?

Here are different ways to perform method overloading:

1. Overloading by changing number of arguments

```
class MethodOverloading {  
    private static void display(int a){  
        System.out.println("Arguments: " + a);  
    }  
  
    private static void display(int a, int b){  
        System.out.println("Arguments: " + a + " and " + b);  
    }  
  
    public static void main(String[] args) {  
        display(1);  
        display(1, 4);  
    }  
}
```

When you run the program, the output will be:

Arguments: 1

Arguments: 1 and 4

2. By changing the datatype of parameters

```
class MethodOverloading {  
  
    // this method accepts int  
    private static void display(int a){  
        System.out.println("Got Integer data.");  
    }  
  
    // this method  accepts String object  
    private static void display(String a){  
        System.out.println("Got String object.");  
    }  
  
    public static void main(String[] args) {  
        display(1);  
        display("Hello");  
    }  
}
```

When you run the program, the output will be:

Got Integer data.

Got String object.

Here, both overloaded methods accept one argument. However, one accepts argument of type `int` whereas other accepts `String` object.

Let's look at a real world example:

```
class HelperService {  
  
    private String formatNumber(int value) {  
        return String.format("%d", value);  
    }  
  
    private String formatNumber(double value) {  
        return String.format("%.3f", value);  
    }  
  
    private String formatNumber(String value) {  
        return String.format("%.2f", Double.parseDouble(value));  
    }  
  
    public static void main(String[] args) {  
        HelperService hs = new HelperService();  
        System.out.println(hs.formatNumber(500));  
        System.out.println(hs.formatNumber(89.9934));  
        System.out.println(hs.formatNumber("550"));  
    }  
}
```

When you run the program, the output will be:

500

89.993

550.00

In Java, you can also overload constructors in a similar way like methods.

Recommended Reading: [Java Constructor Overloading](#)

Important Points

- Two or more methods can have same name inside the same class if they accept different arguments. This feature is known as method overloading.
- Method overloading is achieved by either:
 - changing the number of arguments.
 - or changing the datatype of arguments.
- Method overloading is not possible by changing the return type of methods.

What is a Constructor?

A constructor is similar to a method (but not actually a method) that is invoked automatically when an object is instantiated.

Java compiler distinguish between a [method](#) and a constructor by its name and return type. In Java, a constructor has same name as that of the class, and doesn't return any value.

```
class Test {  
  
    Test() {  
  
        // constructor body  
  
    }  
  
}
```

Here, `Test()` is a constructor; it has same name as that of the class and doesn't have a return type.

```
class Test {  
  
    void Test() {  
  
        // method body  
  
    }  
  
}
```

Here, `Test()` has same name as that of the class. However, it has a return type `void`. Hence, it's a method not a constructor.

Recommended Reading: [Why do constructors not return values?](#)

Example: Java Constructor

```
class ConsMain {  
    private int x;  
  
    // constructor  
    private ConsMain(){  
        System.out.println("Constructor Called");  
        x = 5;  
    }  
  
    public static void main(String[] args){  
        ConsMain obj = new ConsMain();  
        System.out.println("Value of x = " + obj.x);  
    }  
}
```



```
}
```

When you run the program, the output will be:

```
Constructor Called
```

```
Value of x = 5
```

Here, `ConsMain()` constructor is called when `obj` object is instantiated.

A constructor may or may not accept arguments.

No-Arg Constructor

If a Java constructor does not accept any parameters, it is a no-arg constructor. Its syntax is:

```
accessModifier ClassName() {  
  
    // constructor body  
  
}
```

Example of no-arg constructor

```
class NoArgCtor {  
  
    int i;  
  
    // constructor with no parameter  
    private NoArgCtor(){  
        i = 5;  
    }  
}
```

```

        System.out.println("Object created and i = " + i);
    }

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

```

When you run the program, the output will be:

```
Object created and i = 5
```

Here, `NoArgCtor()` constructor doesn't accept any parameters.

Did you notice that the access modifier of `NoArgCtor()` constructor is private?

This is because the object is instantiated from within the same class. Hence, it can access the constructor.

However, if the object was created outside of the class, you have to declare the constructor `public` to access it. For example:

```

class Company {
    String domainName;
    // object is created in another class
    public Company(){
        domainName = "programiz.com";
    }
}

public class CompanyImplementation {

    public static void main(String[] args) {

```

```
Company companyObj = new Company();
System.out.println("Domain name = "+ companyObj.domainName);
}
}
```

When you run the program, the output will be:

```
Domain name = programiz.com
```

Default Constructor

If you do not create constructors yourself, the Java compiler will automatically create a no-argument constructor during run-time. This constructor is known as default constructor. The default constructor initializes any uninitialized instance variables.

Type	Default Value
boolean	false
byte	0
short	0
int	0
long	0L
char	\u0000
float	0.0f
double	0.0d
object	Reference null

Example: Default Constructor

```
class DefaultConstructor {  
  
    int a;  
    boolean b;  
  
    public static void main(String[] args) {  
  
        DefaultConstructor obj = new DefaultConstructor();  
  
        System.out.println("a = " + obj.a);  
        System.out.println("b = " + obj.b);  
    }  
}
```

The above program is equivalent to:

```
class DefaultConstructor {  
  
    int a;  
    boolean b;  
  
    private DefaultConstructor() {  
        a = 0;  
        b = false;  
    }  
  
    public static void main(String[] args) {  
  
        DefaultConstructor obj = new DefaultConstructor();  
  
    }  
}
```

```
        System.out.println("a = " + obj.a);
        System.out.println("b = " + obj.b);
    }
}
```

Recommended Reading: *Java Visibility Modifiers*

Parameterized Constructor

A constructor may also accept parameters. It's syntax is:

```
accessModifier ClassName(arg1, arg2, ..., argn) {

    // constructor body

}
```

Example: Parameterized constructor

```
class Vehicle {

    int wheels;

    private Vehicle(int wheels){
        wheels = wheels;
        System.out.println(wheels + " wheeler vehicle created.");
    }

    public static void main(String[] args) {
        Vehicle v1 = new Vehicle(2);
        Vehicle v2 = new Vehicle(3);
    }
}
```

```
        Vehicle v3 = new Vehicle(4);  
    }  
}
```

When you run the program, the output will be:

```
2 wheeler vehicle created.
```

```
3 wheeler vehicle created.
```

```
4 wheeler vehicle created.
```

Here, we have passed an argument of type `int` (number of wheels) to the constructor during object instantiation.

Constructors Overloading in Java

Similar like [method overloading](#), you can also overload constructors if two or more constructors are different in parameters. For example:

```
class Company {  
  
    String domainName;  
  
    public Company(){  
        this.domainName = "default";  
    }  
  
    public Company(String domainName){  
        this.domainName = domainName;  
    }  
}
```



```

public void getName(){
    System.out.println(this.domainName);
}

public static void main(String[] args) {
    Company defaultObj = new Company();
    Company programizObj = new Company("programiz.com");

    defaultObj.getName();
    programizObj.getName();
}
}

```

When you run the program, the output will be:

default

programiz.com

Recommended Reading: [this keyword in Java](#)

- Constructors are invoked implicitly when you instantiate objects.
- The two rules for creating a constructor are:
 - A Java constructor name must exactly match with the class name (including case).
 - A Java constructor must not have a return type.
- If a class doesn't have a constructor, Java compiler automatically creates a default constructor during run-time. The default constructor initialize instance variables with default values. For example: `int` variable will be initialized to 0
- Constructor types:
 - No-Arg Constructor - a constructor that does not accept any arguments
 - Default Constructor - a constructor that is automatically created by the Java compiler if it is not explicitly defined.
 - Parameterized constructor - used to specify specific values of variables in object
- Constructors cannot be abstract or static or final.
- Constructor can be overloaded but can not be overridden.

Java static keyword

The **static keyword** in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

1) Java static variable

If you declare any variable as static, it is known static variable.

- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees, college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable

It makes your program **memory efficient** (i.e it saves memory).

Understanding problem without static variable

```
1. class Student{  
2.     int rollno;  
3.     String name;  
4.     String college="ITS";  
5. }
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created. All student have its unique rollno and name so instance data member is good. Here, college refers to the common property of all objects. If we make it static, this field will get memory only once.

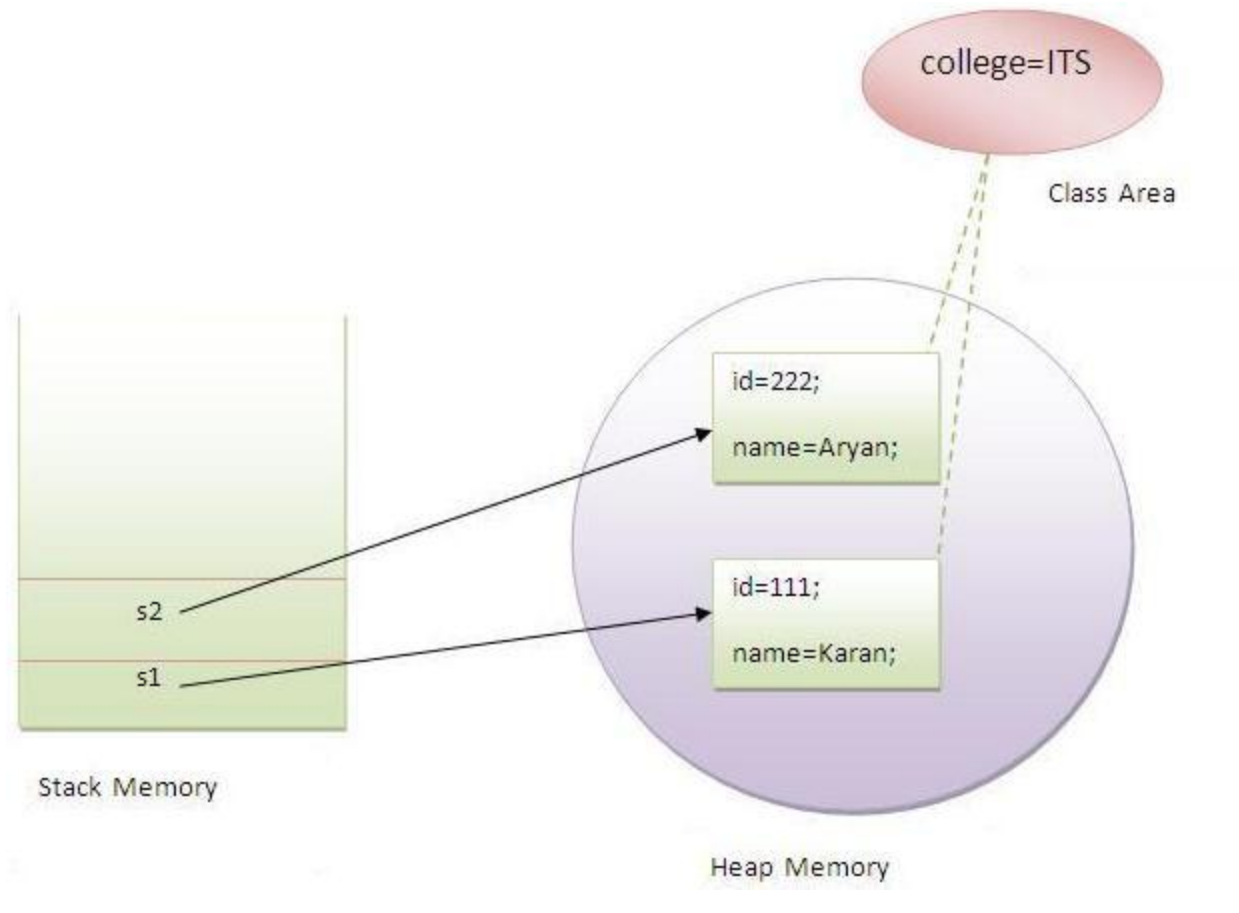
Java static property is shared to all objects.

Example of static variable

```
1. //Program of static variable
2.
3. class Student8{
4.     int rollNo;
5.     String name;
6.     static String college ="ITS";
7.
8.     Student8(int r,String n){
9.         rollNo = r;
10.        name = n;
11.    }
12.    void display (){System.out.println(rollNo+" "+name+" "+college);}
13.
14.    public static void main(String args[]){
15.        Student8 s1 = new Student8(111,"Karan");
16.        Student8 s2 = new Student8(222,"Aryan");
17.
18.        s1.display();
19.        s2.display();
20.    }
21. }
```

Test it Now

```
Output:111 Karan ITS
        222 Aryan ITS
```

Program of counter without static variable

In this example, we have created an instance variable named `count` which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable, if it is incremented, it won't reflect to other objects. So each objects will have the value 1 in the `count` variable.

```
1. class Counter{
2.     int count=0;//will get memory when instance is created
3.
4.     Counter(){
5.         count++;
6.         System.out.println(count);
7.     }
8.
9.     public static void main(String args[]){
10.
11.         Counter c1=new Counter();
```

```
12. Counter c2=new Counter();
13. Counter c3=new Counter();
14.
15. }
16. }
```

Test it Now

```
Output:1
      1
      1
```

Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
1. class Counter2{
2.   static int count=0;//will get memory only once and retain its value
3.
4.   Counter2(){
5.     count++;
6.     System.out.println(count);
7.   }
8.
9.   public static void main(String args[]){
10.
11.   Counter2 c1=new Counter2();
12.   Counter2 c2=new Counter2();
13.   Counter2 c3=new Counter2();
14.
15. }
16. }
```

Test it Now

```
Output:1
      2
      3
```

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

Example of static method

```

1. //Program of changing the common property of all objects(static field).
2.
3. class Student9{
4.     int rollno;
5.     String name;
6.     static String college = "ITS";
7.
8.     static void change(){
9.         college = "BBDIT";
10.    }
11.
12.    Student9(int r, String n){
13.        rollno = r;
14.        name = n;
15.    }
16.
17.    void display (){System.out.println(rollno+" "+name+" "+college);}
18.
19.    public static void main(String args[]){
20.        Student9.change();
21.
22.        Student9 s1 = new Student9 (111,"Karan");
23.        Student9 s2 = new Student9 (222,"Aryan");
24.        Student9 s3 = new Student9 (333,"Sonoo");
25.
26.        s1.display();
27.        s2.display();
28.        s3.display();
29.    }
30. }

```

Test it Now


```
Output:111 Karan BBDIT
        222 Aryan BBDIT
        333 Sonoo BBDIT
```

Another example of static method that performs normal calculation

```
1. //Program to get cube of a given number by static method
2.
3. class Calculate{
4.     static int cube(int x){
5.         return x*x*x;
6.     }
7.
8.     public static void main(String args[]){
9.         int result=Calculate.cube(5);
10.        System.out.println(result);
11.    }
12.}
```

Test it Now

```
Output:125
```

Restrictions for static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

```
1. class A{
2.     int a=40;//non static
3.
4.     public static void main(String args[]){
5.         System.out.println(a);
6.     }
7. }
```

Test it Now

```
Output:Compile Time Error
```

Q) why java main method is static?

Ans) because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation.

3) Java static block

- Is used to initialize the static data member.
- It is executed before main method at the time of classloading.

Example of static block

```
1. class A2{
2.     static{System.out.println("static block is invoked");}
3.     public static void main(String args[]){
4.         System.out.println("Hello main");
5.     }
6. }
```

Test it Now

```
Output:static block is invoked
        Hello main
```

Introduction to Object Oriented Programming

Object-oriented Programming is widely used concept in modern programming languages such as C++,Java, Perl and Python.

- Its programming style is associated with concepts of class and objects and various concepts like Inheritance, encapsulation ,Abstraction and polymorphism.

Why Object Oriented Programming?

prior to object-oriented programming (OOP), programs were written using procedural languages. Procedural languages stress functions. The bigger problems are broken down into smaller sub-problems and written as functions.

- Procedural languages did not pay attention to data. As a result , data was almost neglected, data security was easily compromised.
- Examples of procedural languages include Fortran, COBOL and C, which have been around since the 1960s and 70s.

Procedure oriented Programming(POP)

- Conventional programming languages such as Cobol, Fortran and c is commonly known as procedural oriented programming languages.

Some characteristics exhibited by POP are: 1.Emphasis is on doing things

2.Large programs are divided into smaller programs known as functions.

3.Data moves around the system from function to function. *Limitations of POP are:*

1.Emphasis is on function rather than on data . Any function in program can access , modify data and there is no security for it

2.Code reusability is not provided.

Object oriented programming Paradigm

- OOP is developed by retaining all the best features of structured programming method/procedural method, to which they have added many concepts which facilitates efficient programming.
- OOP treat data as critical element in the program development and does not allow it move freely around the system. It ties data more closely to the function that operate on it and protects it from accidental modifications from outside functions.
- OOP allows decomposition of a problem into number of entities called objects and then builds data and functions around these objects
- Data of an object can be accessed only by the functions associated with object. However functions of one object can access the functions of other objects
- Objects may communicate with each other through functions

Difference between OOP(Object oriented programming)and POP(procedure oriented programming):

OOP	POP
➤ Object oriented.	❓Structure oriented.
➤ Program is divided into objects.	❓Program is divided into functions.
➤ Bottom-up approach.	❓Top-down approach.
➤ Emphasis is on data	❓Emphasis is on function
➤ It uses access specifier. Data is security for data	❓It doesn't use access specifier. No highly secure
➤ Encapsulation is used to hide the data.	❓No data hiding.
➤ Inheritance concept in OOP facilitates reusability of existing	❓No facility of reusability for existing programs programs
➤ C++, Java.	❓C, Pascal.

Basic concepts of object oriented programming:

- In Object oriented programming we write programs using classes and objects utilizing features of OOPs such as
1. Objects• Objects are the basic run-time entities in object- oriented system. An Object is an entity that has state, behaviour and identity. There are many objects around us.
 - E.g. A computer mouse, is an object. It is considered an object with state and behaviour. Its states would be its colour, size and brand name and its behaviour would be left-click, right- click.
 2. Classes• A class is an entity that helps the programmer to define a new complex data type. Objects are the variables of type class. A class defines the data and behaviour of objects. In simple words, A class is a collection of objects of similar type.
 - E.g. mango, apple and orange are members of the class fruit.

Class and Objects

- A class is like a blueprint of data member and functions
- and object is an instance of class.

3. Data Abstraction• Data Abstraction refers to the act of re- presenting essential features without including the back-ground details. It is concerned with separating the behaviour of a data object from its re-presentation.
 - E.g. Executable file of a program.

4. Encapsulation • The process of binding data members and functions in a class is known as, encapsulation. Encapsulation is the powerful feature (concept) of object-oriented programming. With the help of this concept, data is not accessible to the outside world and only those functions which are declared in the class, can access it.

5. Data Hiding • Data Hiding is similar to encapsulation. Basically, encapsulating data members and functions in a class promotes data hiding. This concept will help us to provide the essential features to the users and hide the details. In short, encapsulating through private access modifier (label) is known as data hiding.

6. Inheritance • Inheritance is a process by which objects of new class acquire the properties of objects of existing (base) class. It is in hierarchical order. The concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it.

7. Polymorphism • Polymorphism is an important object-oriented programming concept. This is a Greek term, means the ability to take more than one form. The process of using a single function name to perform different types of tasks is known as function-overloading.

8. Binding • Binding refers to the linking of a procedure call to the code (its body) to be executed in response to the call.

9. Message Passing: objects communicate with one-another by sending and receiving information much the same way as people send messages to one- another.

Introduction to C++

- C++ language is a direct descendant of C programming language with additional features such as object oriented programming, exception handling etc.
- C++ is developed by Bjarne Stroustrup starting in 1979 at Bell Labs. C++ runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.
- C++ is general purpose , high level ,compiler based object oriented programming language

Some of the *features* are as follows:

- Simple: It is a simple language in the sense that programs can be broken down into logical units and parts, has a rich library support and a variety of data-types.
- Machine Independent but Platform Dependent: A C++ executable is not platform-independent (compiled programs on Linux won't run on Windows), however they are machine independent.
- Mid-level language: It is a mid-level language as we can do both systems-programming (drivers, kernels, networking etc.) and build large-scale user applications (Media Players, Photoshop, Game Engines etc.)
- Rich library support:
- Speed of execution: C++ programs excel in execution speed. Since, it is a compiled language, and also hugely procedural..
- Pointer and direct Memory-Access: C++ provides pointer support which aids users to directly manipulate storage address.
- Object-Oriented: Object-Oriented support helps C++ to make maintainable and extensible programs. i.e. Large-scale applications can be built. Procedural code becomes difficult to maintain as code-size grows.
- Compiled Language: C++ is a compiled language, contributing to its speed.

Applications of C++:

C++ finds varied usage in applications such as:

- Operating Systems & Systems Programming.

e.g. *Linux-based OS (Ubuntu etc.)*

- Graphics & Game engines (*Photoshop, Blender, Unreal-Engine*)

- Database Engines (*MySQL, MongoDB, Redis etc.*)

First C++ Program

```
/*Multiple line comment
*/
#include<iostream>

using namespace std;

//This is where the execution of program begins int
main()
{
    // displays Hello World! on screen
    cout<<"Hello World!";
    return 0;
}
```

Output:

Hello World!

Input and Output operators:

Output Operator:

The cout is a predefined object of ostream class in iostream header file. It is connected with the standard output device, which is usually a display screen. This object can also display the value of variables on screen. The cout is used in conjunction with stream insertion operator (<<) to display the output on a console

Input Operator:

The cin is a predefined object of istream class. It is connected with the standard input device, which is usually a keyboard. The cin is used in conjunction with stream extraction operator (>>) to read the input from a console.

```
#include <iostream>
using namespace std;
int main( ) {
    int age;
    cout << "Enter your age: "; cin >> age;
    cout << "Your age is: " << age << endl;
}
```

Output:Enter your age: 22 Your age is: 22

The endl is a predefined object of ostream class. It is used to insert a new line characters

```
#include <iostream>
using namespace std;
int main( )
{
    cout << "C++ Tutorial";
    cout << " Javatpoint"<<endl;
    cout << "End of line"<<endl;
}
```

Output:

C++ Tutorial Javatpoint
End of line

`<iostream>`

It is used to define the `cout`, `cin` and `cerr` objects, which correspond to standard output stream, standard input stream and standard error stream, respectively.

Namespace

- Namespace defines scope for the identifiers that are used in a program. For using the identifiers defined in namespace scope we must include the following directive , like `using namespace std;`
- Here , `std` is the namespace where ANSI C++ standard class libraries are defined. This will bring all the identifiers defined in `std` to the current global scope.
- `using` and `namespace` are the keywords of c++

C++ Programming Fundamentals

C++ Identifiers

- A C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore (`_`) followed by zero or more letters, underscores, and digits (0 to 9).
- Here are some examples of acceptable identifiers

Mohd abc move_name a_123 myname5 _temp j a23b9 retVal

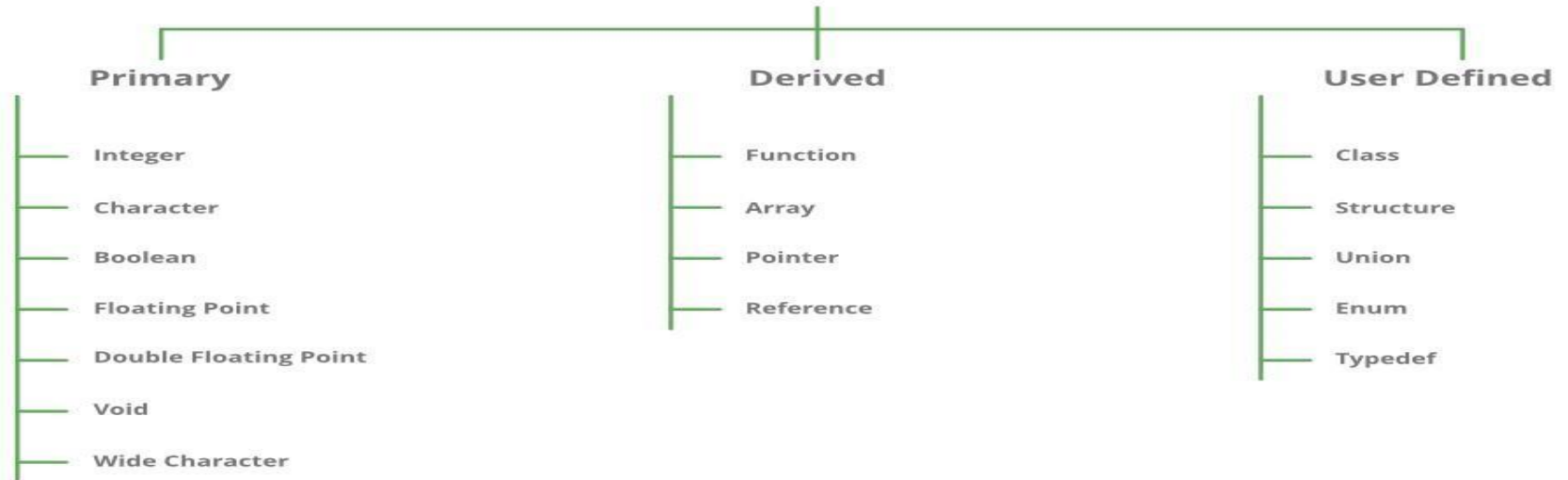
C++ Keywords

The following list shows the reserved words in C++. These reserved words may not be used as constant or variable or any other identifier names.

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try

case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

DataTypes in C / C++




```
//program demonstrating data types
```

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    int myNum = 5;          // Integer (whole number) float
```

```
    myFloatNum = 5.99; // Floating point number double
```

```
    myDoubleNum = 9.98; // Floating point number
```

```
    char myLetter = 'D';    // Character bool
```

```
    myBoolean = true;      // Boolean string
```

```
    myString = "Hello"; // String
```

```
    // Print variable values
```

```
    cout << "int: " << myNum << "\n";
```

```
    cout << "float: " << myFloatNum << "\n";
```

```
    cout << "double: " << myDoubleNum << "\n";
```

```
    cout << "char: " << myLetter << "\n";
```

```
    cout << "bool: " << myBoolean << "\n";  
    cout << "string: " << myString << "\n";  
    return 0;  
}
```

Output:

int: 5 float:

5.99

double:

9.98 char:

D bool: 1

string:

Hello

Operators in c++

An operator is simply a symbol that is used to perform operations. There are following types of operators to perform different types of

	Operator	Type
Binary Operator	+, -, *, /, %	Arithmetic Operators
	<, <=, >, >=, ==, !=	Relational Operators
	&&, , !	Logical Operators
	&, , <<, >>, ~, ^	Bitwise Operators
	=, +=, -=, *=, /=, %=	Assignment Operators
Unary Operator	++, --	Unary Operator
Ternary Operator	?:	Ternary or Conditional Operator

- All C operators are valid in c++ also. In addition, c++ introduces some new operators such as insertion operator << and extraction operator >>

Other new operators are:

- 1) • :: scope resolution operator
- 2) • ->* pointer to member declarator
- 3) • ::* pointer to member declarator
- 4) • .* pointer to member declarator
- 5) • delete memory release operator
- 6) • endl line feed operator
- 7) • new Memory allocation operator
- 8) • setw Field width operator

scope resolution operator is ::. It is used for following purposes.

1. To access a global variable when there is a local variable with same name:

```
#include<iostream>
using namespace std;
int x; // Global x
```



```
int main()
{
    int x = 10; // Local x
    cout << "Value of global x is " << ::x;
    cout << "\nValue of local x is " << x; return
    0;
}
```

Output:Value of global x is 0
Value of local x is 10

2) For namespace we can use the namespace name with the scope resolution operator to refer that class without any conflicts

// Use of scope resolution operator for namespace.

```
#include<iostream>
int main()
{
    std::cout << "Hello" << std::endl;
}
```

Here cout and endl belong to std namespace

3) To define a function outside a class.

Type Definition typedef keyword is used to assign a new name to any existing data-type.

Following is the syntax of typedef
typedef current_name new_name;

```
#include <iostream>
using namespace std;
int main()
{
    typedef int marks;
    marks i = 5, j = 8;
    cout << "i = " << i << endl;
    cout << "j = " << j << endl;
    return 0;
}
```

Enumerated Data type

- enum in C++ is a data type that contains fixed set of constants.

```
#include <iostream>
using namespace std;
enum week { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
```

```
int main()
{
    week day; day = Friday;
    cout << "Day: " << day+1<<endl;
    return 0;
}
```

Output:

Day:5

C++ Explicit Conversion

When the user manually changes data from one type to another, this is known as explicit conversion. This type of conversion is also known as type casting.

```
#include <iostream>
using namespace std;
int main() {
    double num_double = 3.56;
    cout << "num_double = " << num_double << endl;

    int num_int1 = (int)num_double;
    cout << "num_int1 = " << num_int1 << endl;
    return 0;
}
```

Output:

num_double=3.56 num_int1=3

Introduction to C++ Classes and Objects

- The classes are the most important feature of C++ that leads to Object Oriented programming.
- Class is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating instance (objects) of that class.

The general form of class definition:

```
class classname
{
private:
    variable declaration;
    function declaration;
public:
    variable declaration;
    function declaration;
};
```


- In C++ a class is defined by using the keyword class followed by the class name.
- The variables inside class definition are called as data members and the functions are called member functions.
- The body of class contains the declaration of variables and functions which are collectively called class members.

For example:

```
class student
{
private:
int roll_number;
public:
char name[10];
void fun1() ;
};
```

- As seen in above example roll_number and name are data members and fun1() is the member function of class name student.
- The keywords private and public are followed by a colon. The members that have been declared as private can be accessed only within the class . on the other hand, public members can be accessed from outside the class also. The data hiding(using private declaration) is the key feature of object oriented programming.
- The use of keyword private is optional because by default , the members of a class are private

Creating Objects

Object is an instance of a class. All the members of the class can be accessed through object.

Syntax to Define Object in C++

className objectVariableName;

Example : student s1; //creating an object of Student

- In this example, Student is the type and s1 is the reference variable that refers to the instance of Student class.
- When class is defined, only the specification for the object is defined; no memory or storage is allocated.
- You can access the data members and member functions by using a . (dot) operator. For example,
- s1.roll_number=10;

//C++ program to demonstrate the use of object and class

```
#include<iostream>
using namespace std;
class MyClass {    // The class
public:           // Access specifier
    int Num1;     // data member
    float Num2 ; //data member
};
int main() {
    MyClass myObj; // Create an object of MyClass

    // Access attributes and set values
    myObj.Num1 = 15;
```

```
myObj.Num2 = 28;
```

```
// Print attribute values  
cout << myObj.Num1<<endl;  
cout << myObj.Num2; return 0;  
}
```

Accessing class members

The private data of a class can be accessed only through member functions of that class.

Format of calling member function:

`Objectname.functionname(actual arguments);`

Example:

```
s1.fun(10,"sam");
```

Member function can be invoked only using object `fun(10,"sam");` is invalid

Similarly `s1.roll_number=10;` is invalid

- Although `s1` is an object of type `student` to which `number` belongs, the `number` (declared private) can be accessed only through a member function and not by the object directly.
- A variable declared as public can be accessed by the objects directly

A variable declared as public can be accessed by the objects directly.

```
#include<iostream>
```

```

using namespace std;
class xyz
{
int x;
public: int
z;
};
int main()
{
xyz ob;
//ob.x=0;//error    x    is    private
ob.z=10;//z        is        public
//cout<<ob.x<<endl; //error
cout<<ob.z;
return 0;
}

```

Defining Member functions

Member functions can be defined in two places:

- Outside the class definition
- Inside the class definition

Outside the class definition

Member functions that are declared inside a class have to be defined separately outside the class.

General form of a member function definition

```
returntype classname::function(argumentlist)
{
    Function body
}
```

class name:: tells the compiler that function functionname belongs to class classname. That is, scope of the function is restricted to the class name specified

```
#include<iostream>
using namespace std;
class person
{
    char
    name[30];
    int age;
    public:
        void getdata(void);
        void display(void);
};
void person::getdata(void)
{
```

```

        cout<<"Enter
        name:"; cin>>name;
        cout<<"Enter age:";
        cin>>age;
    }
    void person::display(void)
    {
        cout<<"\n name:"<<name;
        cout<<"\n Age:"<<age;
    }
    int main()
    {
        person p;

        p.getdata();

        p.display();
        return 0;
    }

```

Output:

Enter name: john

Age:17

Name: john

Age:17

Inside the class definition

- Another method of defining a member function is to replace the function declaration by the actual function definition inside the class.

```
#include<iostream>

using namespace std;
class person
{
    char
    name[30];
    int age;
    public:

        void getdata(void)
        {
            cout<<"Enter
            name:"; cin>>name;
            cout<<"Enter age:";
            cin>>age;

        } void
        display(void)
        {
            cout<<"\n name:"<<name;
            cout<<"\n Age:"<<age;
```

```
}  
};
```

```
int main()  
{  
    person p  
    p.getdata();  
    p.display();  
    return 0;  
}
```


Basic structure of c++ program

1. Documentation Section
2. Preprocessor Directives or Compiler Directives Section (i) Link Section
(ii) Definition Section
3. Global Declaration Section
4. Class declaration or definition
5. Main C++ program function called main () 6. Beginning of the
program: Left brace {
(i) Object declaration part;
(ii) Accessing member functions (using dot operator); 7. End of the main
program: Right brace }

/* C++ program to create a simple class and object. defining member function inside the class*/

```
#include <iostream> using  
namespace std;
```

```
class Hello {  
    public:  
        void sayHello()  
        {  
            cout << "Hello World" << endl;  
        }  
};
```

```
int main() {  
    Hello h;  
  
    h.sayHello();  
    return 0; }
```

output:Hello World

Characteristics of member functions

- A member function can call another member function directly, without using the dot operator.
- Member functions can access the private data of the class. A non-member functions can't do so(except friend function)

Types of variables:

There are three types of variables based on the scope of variables in C++

- Local Variables
- Instance Variables
- Static Variables

A variable provides us with named storage that our programs can manipulate.

Instance variables – Instance variables are declared in a class, but outside a method. When space is allocated for an object in the heap, a slot for each instance variable value is created.

Local variables – Local variables are declared in methods, constructors, or blocks. Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.

Types of variables in C++

```
class GFG {  
    public:  
        static int a; — Static Variable  
        int b; — Instance Variable  
    public:  
        func()  
        {  
            int c; — Local Variable  
        }  
};
```



Access Specifiers in C++

Access specifiers in C++ define how the members of the class can be accessed. C++ has 3 new keywords introduced, namely.

- public
- private
- protected

public

Data members or Member functions which are declared as public can be accessed anywhere in the program (within the same class, or outside of the class).

protected

Data members or Member functions which are declared as protected can be accessed in the derived class or within the same class. private

Data members or Member functions which are declared as private can be accessed within the same class only i.e. the private Data members or Member functions can be accessed within the public member functions of the same class.

```
/* program demonstrating on data hiding*/
```

```
#include<iostream> using  
namespace std; class dates
```

```
{  
  
private:  
  
    int date,month;  
public:  
int year;  
  
};  
  
int main()  
  
{  
  
    dates date1;  
  
    cout<<"program starts"<<endl;  
    date1.year=2020;  
  
    cout<<"now we are in the year"<<date1.year<<"AD"<<endl;  
  
    date1.date=10;//error date is not accessible  
    cout<<date1.date;//error return 0;  
  
}
```

Output: program starts

Now we are in the year 2020 AD

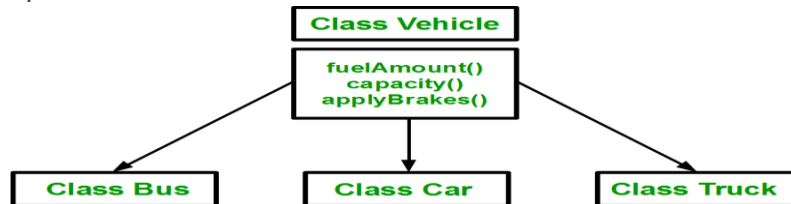
Inheritance

- Inheritance is the process by which the objects of one class can acquire the properties of another class.
- The concept of inheritance provides the idea of code reusability
- This means that we can add additional features to a existing classes without modifying it
- This is possible by deriving new class from existing class
- The existing class is known as the base class(or super class or parent class) and the new class is called as a (derived class or sub class or child class).
- The derived class inherits some or all features of the base class
- Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

Super Class:The class whose properties are inherited by sub class is called Base Class or Super class.

- The main advantage of the inheritance are:
 - ✓ Resuability of code
 - ✓ To increase the reliability of the code
 - ✓ To add some enhancement of base class

Example:



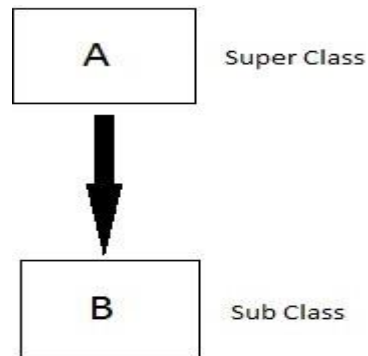
Types Of Inheritance

C++ supports five types of inheritance:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance

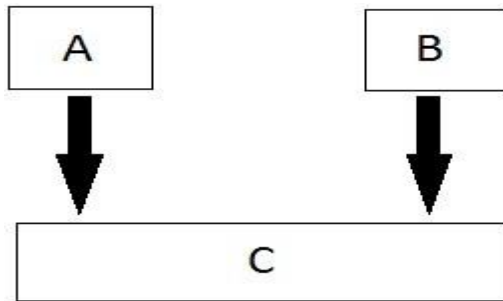
Single Inheritance in C++

In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



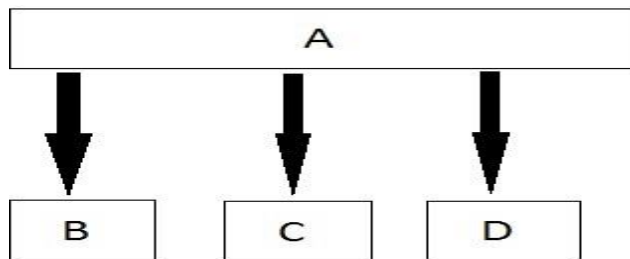
Multiple Inheritance in C++

In this type of inheritance a single derived class may inherit from two or more than two base classes.



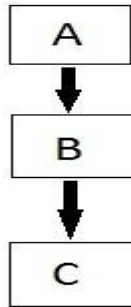
Hierarchical Inheritance in C++

In this type of inheritance, multiple derived classes inherit from a single base class.



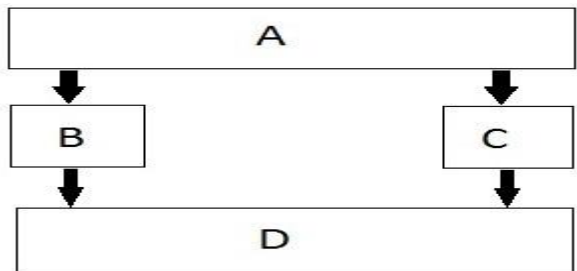
Multilevel Inheritance in C++

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



Hybrid (Virtual) Inheritance in C++

Hybrid Inheritance is combination of Hierarchical and Multilevel Inheritance.



Defining Derived Class

- A derived class is defined by specifying its relationship with the base class in addition to its own details

The general form of defining a derived class is:

```
class derivedclassname : visibilitymode baseclassname
{
//members of derived class
}
```

- The colon(:) indicates that the derived class name is derived from the base class name
- The visibility mode(access specifier) specifies whether the features of the base class are privately derived or publicly derived. The default visibility is private.

Examples:

```
class A
{
members of A
}
class B:private A//private derivation
{
members of B
}
```

```
class A
{
members of A
}
class B:public A//public derivation
{
members of B
}
```

```
//program demonstrating single inheritance
#include <iostream>
using namespace std;
```

```
class x
{
    private:
        int id_p;
};
```

```
class y : public x
{
    public:
        int id_c;
};
```

```
int main()
{
    y obj1;
```

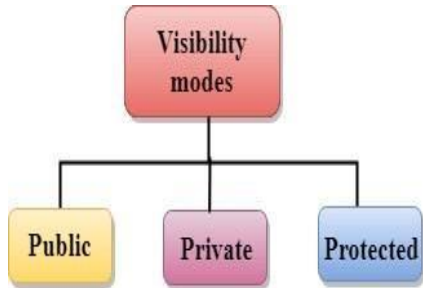
```
    obj1.id_c = 7;  
    obj1.id_p = 91;  
    cout << "Child id is " << obj1.id_c << endl;  
    cout << "Parent id is " << obj1.id_p << endl;  
  
    return 0;  
}
```

Output:

child is 7

Parent is 91

Visibility modes(Access specifiers) can be classified into three categories:



- ✓ •Public: When the member is declared as public, it is accessible to all the functions of the program.
- ✓ •Private: When the member is declared as private, it is accessible within the class only.
- ✓ •Protected: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

Modes of Inheritance

Public mode: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.

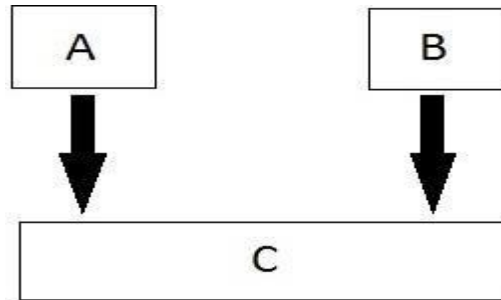
Protected mode: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.

Private mode: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

Base class visibility	Derived class visibility		
	Public	Private	Protected
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Multiple inheritance

- In multiple inheritance, a class can inherit from more than one classes. In simple words, a class can have more than one parent classes.
- Suppose we have to make two classes A and B as the parent classes of class C, then we have to define class C as follows.



- General form of multiple inheritance

- class C: public A, public B

```
{  
    // code  
};
```

```
#include<iostream>
```

```
using namespace std;
```

```
// Base class
```

```
class MyClass {
```

```
public: void myFunction()
```

```
{
```

```
    cout << "Some content in parent class." ;
```

```
}
```

```
};
```

// Another base class

```
class MyOtherClass
{
public:
    void myOtherFunction()
    {
        cout << "Some content in another class." ;
    }
};
```

// Derived class

```
class MyChildClass: public MyClass, public MyOtherClass {
};
```

```
int main()
{
    MyChildClass myObj;
    myObj.myFunction();
    myObj.myOtherFunction();
    return 0;
}
```

Output:

Some content in parent class.some content in another class

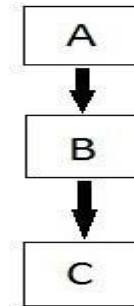
Multilevel Inheritance in C++

- In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class.
- General form of multi level inheritance:

```
class A
{
};
class B:public A
{
};
class C:public B
{
};
```

```
#include <iostream> using
namespace std;
```

```
class A
{ public: void
  display()
  {
    cout<<"Base class content.";
  }
};
class B : public A
{
};
```




```
class C : public B
```

```
{ };
```

```
int main()
```

```
{
```

```
  C obj;
```

```
  obj.display();
```

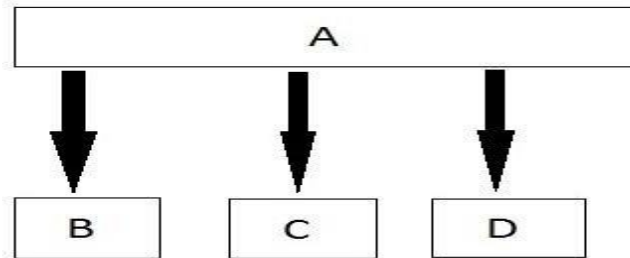
```
  return 0;
```

```
}
```

Output:Base class content

Hierarchical Inheritance

In this type of inheritance, multiple derived classes inherits from a single base class.



Hybrid Inheritance

```
// hierarchial inheritance.cpp
#include <iostream>
using namespace std;
class A //single base class
{

    public:

        int x, y;

        void getdata()

        {

            cout << "\nEnter value of x and y:\n"; cin >> x >> y;

        }

};

class B : public A //B is derived from class base

{

    public:
```

```
void product()
```

```
{
```

```
    cout << "\nProduct= " << x * y;
```

```
}
```

```
};
```

```
class C : public A //C is also derived from class base
```

```
{
```

```
    public:
```

```
        void sum()
```

```
        {
```

```
            cout << "\nSum= " << x + y;
```

```
        }
```

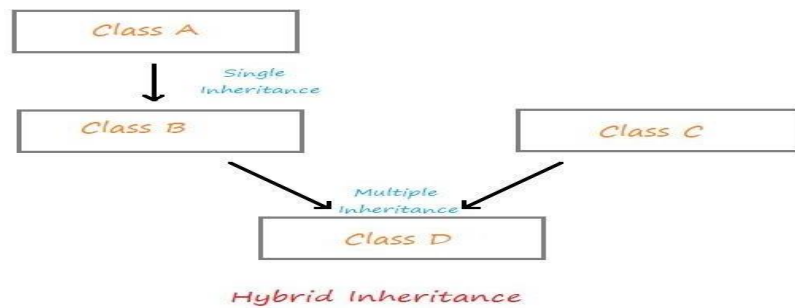
```
};
```

```
int main()
```

```
{
```

```
B obj1;    //object of derived class B
C obj2;    //object of derived class C
obj1.getdata();
obj1.product();
obj2.getdata();
obj2.sum();
return 0;
} //end of program
```

- Hybrid Inheritance is the combination of two or more inheritances : single, multiple , multilevel or hierarchical Inheritances.




```
//hybrid inheritance
#include <iostream>
using namespace std;
class A
{
```

protected:

```
    int a;
    public:
    void get_a()
```

```
{
```

```
    cout << "Enter the value of 'a' : " << endl;
    cin>>a;
}
```

```
};
```

```
class B : public A
```

```
{
```

```
    protected:
    int b;
```

```
public:
```

```
void get_b()
```

```
{
```

```
    cout << "Enter the value of 'b' : " << endl;
```

```
    cin>>b;
```

```
}
```

```
};
```

```
class C
```

```
{
```

```
protected:
```

```
int c;
```

```
public:
```

```
void get_c()
```

```
{
```

```
    cout << "Enter the value of c is : " << endl;
```

```
    cin>>c;
```

```
}
```

```
};
```

```
class D : public B, public C
```

```
{
```

```
    protected:
```

```
    int d;
```

```
    public:
```

```
    void mul()
```

```
    {
```

```
        get_a();
```

```
        get_b();
```

```
        get_c();
```

```
        cout << "Multiplication of a,b,c is : " << a*b*c << endl;
```

```
    }
```

```
};
```

```
int main()
```

```
{
```

```
    D d;
```

```
    d.mul();
```

```
    return 0;
```

```
}
```

Constructors

A constructor in C++ is a special method that is automatically called when an object of a class is created.

➤ How constructors are different from normal member function?

1) A constructor has same name as the class itself.

2) A constructor does not have return type

3) A constructor is automatically called when an object is created Syntax:

```
class_name(parameter1, parameter2, ...)
```

```
{
```

```
// constructor Definition
```

```
}
```

```
//default constructor
```

```
#include<iostream>
```

```
using namespace std;
```

```
class MyClass
```

```
{
```

```
public:
```

```
    MyClass() {
```

```
        cout << "Hello World!";
```

```
    }
```

```
};
```

```
int main() {
```

```
    MyClass myObj; // Create an object of MyClass (this will call the constructor) return 0;
```

```
}
```


Output:Hello World!

```
//parameterized constructors
```

```
#include <iostream>
```

```
using namespace std;
```

```
class Calc
```

```
{
```

```
public:
```

```
    //int val;
```

```
public:
```

```
    Calc(int x)
```

```
{
```

```
    cout << x;
```

```
    }
```

```
};
```

```
int main()
```

```
{
```

```
    Calc c1(10);
```

```
}
```

Output: 10

Types of Constructor in C++

There are two types of constructor in C++.

1) Default constructor 2) Parameterized constructor

1) Default Constructor

A default constructor doesn't have any arguments (or parameters)

2) Parameterized Constructor

Constructors with parameters are known as Parameterized constructors.

These type of constructor allows us to pass arguments while object creation.

What is Destructor?

- Destructor is a member function which destructs or deletes an object.
- Destructors doesn't take any arguments and don't return anything • Destructors have same name as the class preceded by a tilde(~) Syntax:

```
~class_name()
```

```
{
```

```
//Some code
```

```
}
```

```
#include <iostream>
```

```
using namespace std;
```

```
class Calc {
```

```
public:
```

```
int val;
```

```
public:
```

```
Calc()
```

```
{
```

```
val = 20;
```

```
cout << val;
```

```
}
```

```
~Calc()
```

```
{
```

```
cout << "destructor is called";
```

```
}
```

```
};
```

```
int main()
```

```
{
```

```
Calc c1;
```

```
}
```

```
Output:
```

```
20
```

```
destructor is called
```

Function Overloading in C++

- ✓ In a class definition we can have two or more functions having the same name. This is called function overloading.
- ✓ To resolve ambiguity, the number of parameters and/or datatypes in the parameter list of each function must be different. The number of parameters and their data types is called the signature of a function.
- ✓ so even if the names of the functions are same,by their signatures the functions are uniquely identified.
- ✓ Function overloading is usually used to enhance the readability of the program. If you have to perform one single operation but with different number or types of arguments, then you can simply overload the function.

//function overloading

```
#include<iostream>
#define pi 3.14
using namespace std;
class figure
{
public:
    void area(int x,int y)
    {
        cout<<"area of rectangle is:"<<x*y<<endl;
    }

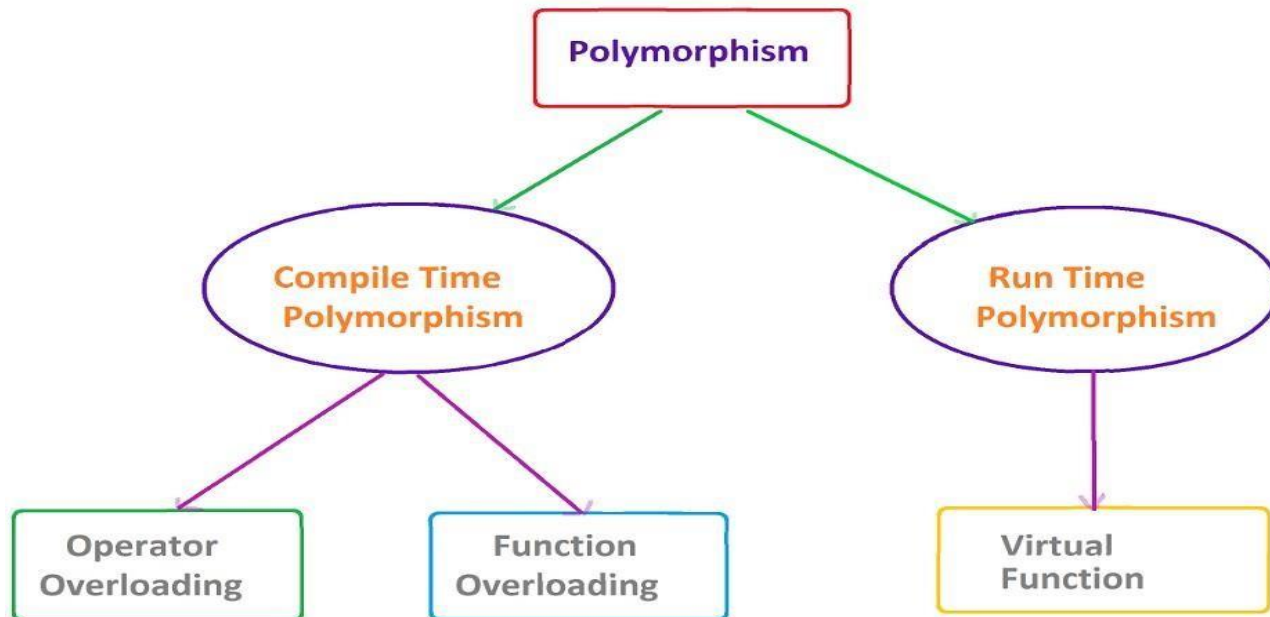
    void area(float r)
    {
        cout<<"area of circle is:"<<pi*r*r<<endl;
    }
    void area(double x,double y)
    {
        cout<<"area of traingle is:"<<(x*y)/2<<endl;
    }
};

int main()
{
    figure geo;
    geo.area(7.0);
    geo.area(2,3);
    geo.area(8.3,4.5);
    return 0;
}
```

Polymorphism

- The word polymorphism means having many forms. Polymorphism is a feature of [OOPs](#) that allows the object to behave differently in different conditions. In C++ we have two types of polymorphism:

- 1) Compile time Polymorphism – This is also known as static (or early) binding.
- 2) Runtime Polymorphism – This is also known as dynamic (or late) binding.



1) Compile time Polymorphism

Function overloading and Operator overloading are perfect example of Compile time polymorphism.

Compile time Polymorphism Example

In this example, we have two functions with same name but different number of arguments. Based on how many parameters we pass during function call determines which function is to be called, this is why it is considered as an example of polymorphism because in different conditions the output is different. Since, the call is determined during compile time thats why it is called compile time polymorphism.

2) Runtime Polymorphism

Function overriding is an example of Runtime polymorphism.

Function Overriding: When child class declares a method, which is already present in the parent class then this is called function overriding, here child class overrides the parent class.

C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type.

- It helps the programmer to use operators with objects of classes. The outcome of operator overloading is that objects can be used in a natural manner as the variables of basic data types.

Defining operator overloading

Class to which operator is applied is called operator function. operator functions must be either member function.

Syntax:

The diagram illustrates the syntax of operator overloading. It shows a code snippet enclosed in a box, with two labels above it: 'Keyword' and 'Operator to be overloaded'. Arrows point from these labels to the corresponding parts of the code snippet.

```
ReturnType classname :: Operator OperatorSymbol (argument list)
{
    \\ Function body
}
```

➤ The process of overloading involves the following steps:

- 1.create a class that defines the data types that is used in overloading operation.
- 2.Declare the operator function operator op() in the public part of class.

- 3. Define the operator function to implement the required operations.

Operator that cannot be overloaded are as follows:

Scope operator (::) Sizeof member selector (.) member

pointer selector (*) ternary operator (?:)

```
//overloading unary operators
```

```
#include<iostream>
```

```
using namespace std;
```

```
class space
```

```
{
```

```
int x;
```

```
int y;
```

```
int z;
```

```
public:
```

```
void getdata(int a,int b,int c);
```

```
void display();
```

```
void operator-();
```

```
};
```

```
void space::getdata(int a,int b,int c)
```

```
{
```

```
x=a;
```

```
y=b;
```

```
z=c;
```

```
}
```

```
void space::display()
```

```
{
```

```
cout<<x<<" ";
```

```
cout<<y<<" ";
```

```
cout<<z<<" ";
```

```
}
```

```
void space::operator-()
```

```

{
x=-x;
y=-y;
z=-z;
}
int main()
{
space s;
s.getdata(10,-20,30);
cout<<"s:";
s.display();
-s;//activate operator -() function
cout<<"s:";
s.display();
return 0;
}

```

Output:

S:10 -20 30

S:-10 20 30

Explanation:

The function operator `-()` takes no argument. the unary minus operator when applied to an object changes the sign of data members of the objects.

Data Abstraction in C++

- Data Abstraction is a process of providing only the essential details to the outside world and hiding the internal details, i.e., representing only the essential details in the program.

Data Abstraction can be achieved in two ways:

Abstraction using classes • Abstraction in header files.

Abstraction using classes: An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.

Abstraction in header files: Another type of abstraction is header file. For example, `pow()` function available is used to calculate the power of a number without actually knowing which algorithm function uses to calculate the power. Thus, we can say that header files hide all the implementation details from the user.

Access Specifiers Implement Abstraction:

Public specifier: When the members are declared as public, members can be accessed anywhere from the program.

Private specifier: When the members are declared as private, members can only be accessed only by the member functions of the class.

Let's see a simple example of abstraction in header files.

Abstract Class and Pure Virtual Function in C++

In [C++](#), we use terms abstract class and interface interchangeably. A class with pure virtual function is known as abstract class. For example the following function is a pure virtual function:

➤ `virtual void fun() = 0;`

A pure virtual function is marked with a virtual keyword and has `= 0` after its signature. You can call this function an abstract function as it has no body. The derived class must give the implementation to all the pure virtual functions of parent class else it will become abstract class by default.

```
//Abstract   baseclass
#include<iostream>
using namespace std;

// Abstract base class
class Base
{
    public:
        virtual void show() = 0;
//Pure Virtual Function
};

void Base :: show()
//Pure Virtual definition
{
    cout << "Pure Virtual
definition\n";
}

class Derived:public Base
{
    public:
        void show()
```



```
{  
    cout <<  
    "Implementation of  
    Virtual Function in  
    Derived class\n";  
}  
};
```

```
int main()  
{  
    Base *b;  
    Derived d;  
    b = &d;  
    b->show();  
}
```

Interfaces

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.

The C++ interfaces are implemented using abstract classes and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

A class is made abstract by declaring at least one of its functions as pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration

Encapsulation In C++

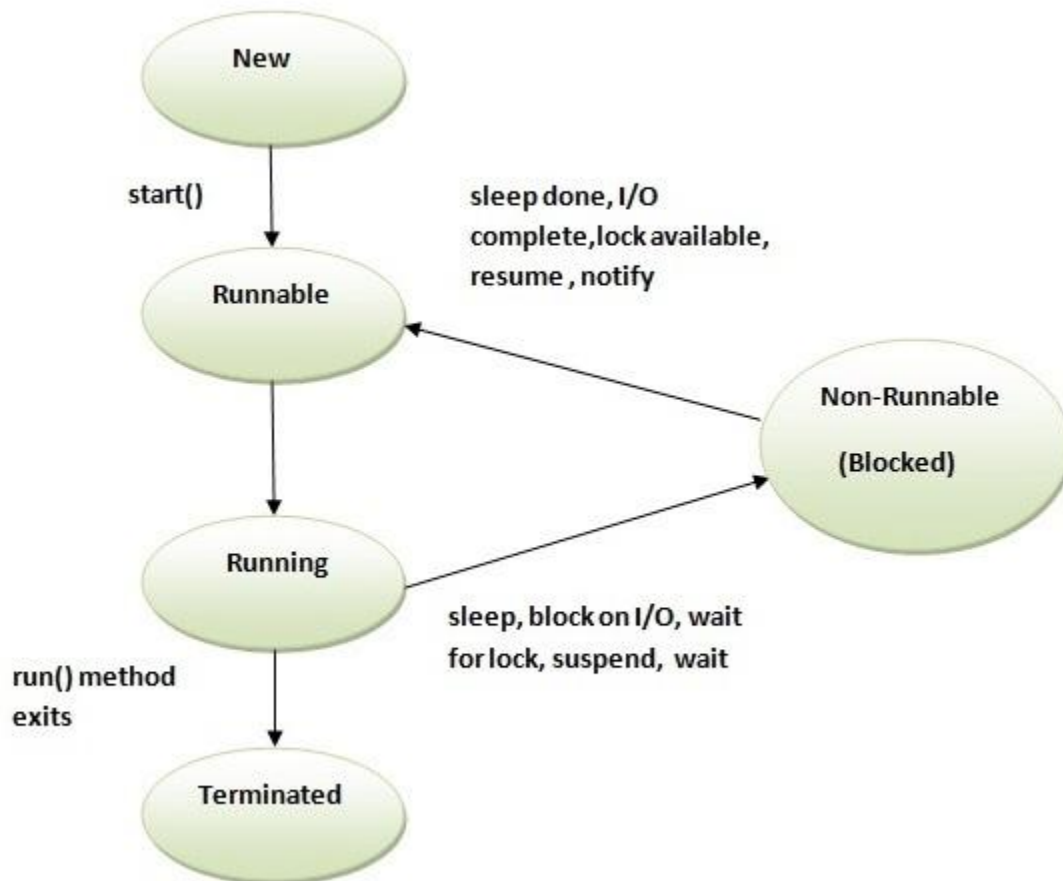
When all the data members and member functions are combined in a single unit called class, this process is called *Encapsulation*. In other words, wrapping the data together and the functions that manipulate them.

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated



1) New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running

The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated

A thread is in terminated or dead state when its run() method exits

How to create thread

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread. JVM calls the run() method on the thread.
3. **public void sleep(long milliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long milliseconds):** waits for a thread to die for the specified milliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(deprecated).
16. **public void resume():** is used to resume the suspended thread(deprecated).
17. **public void stop():** is used to stop the thread(deprecated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run():** is used to perform action for a thread.
-

Starting a thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
 - The thread moves from New state to the Runnable state.
 - When the thread gets a chance to execute, its target run() method will run.
-

1) Java Thread Example by extending Thread class

```
1. class Multi extends Thread{
2. public void run(){
3. System.out.println("thread is running...");
4. }
5. public static void main(String args[]){
6. Multi t1=new Multi();
7. t1.start();
8. }
9. }
```

Output:thread is running...

2) Java Thread Example by implementing Runnable interface

```
1. class Multi3 implements Runnable{
2. public void run(){
3. System.out.println("thread is running...");
4. }
5.
6. public static void main(String args[]){
```

```
7. Multi3 m1=new Multi3();
8. Thread t1 =new Thread(m1);
9. t1.start();
10. }
11. }
```

```
Output:thread is running...
```

If you are not extending the Thread class, your class object would not be treated as a thread object. So you need to explicitly create Thread class object. We are passing the object of your class that implements Runnable so that your class run() method may execute.

Thread Scheduler in Java

Thread scheduler in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Difference between preemptive scheduling and time slicing

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

Sleep method in java

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Syntax of sleep() method in java

The Thread class provides two methods for sleeping a thread:

- `public static void sleep(long milliseconds)throws InterruptedException`
- `public static void sleep(long milliseconds, int nanos)throws InterruptedException`

Example of sleep method in java

```

1. class TestSleepMethod1 extends Thread{
2.     public void run(){
3.         for(int i=1;i<5;i++){
4.             try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
5.             System.out.println(i);
6.         }
7.     }
8.     public static void main(String args[]){
9.         TestSleepMethod1 t1=new TestSleepMethod1();
10.        TestSleepMethod1 t2=new TestSleepMethod1();
11.
12.        t1.start();
13.        t2.start();
14.    }
15.}

```

Output:

```

1
1
2
2
3
3
4
4

```

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time, the thread scheduler picks up another thread and so on.

The join() method

The `join()` method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

Syntax:

```
public void join()throws InterruptedException
```

```
public void join(long milliseconds)throws InterruptedException
```

Example of join() method

```
1. class TestJoinMethod1 extends Thread{
2.     public void run(){
3.         for(int i=1;i<=5;i++){
4.             try{
5.                 Thread.sleep(500);
6.             }catch(Exception e){System.out.println(e);}
7.             System.out.println(i);
8.         }
9.     }
10. public static void main(String args[]){
11.     TestJoinMethod1 t1=new TestJoinMethod1();
12.     TestJoinMethod1 t2=new TestJoinMethod1();
13.     TestJoinMethod1 t3=new TestJoinMethod1();
14.     t1.start();
15.     try{
16.         t1.join();
17.     }catch(Exception e){System.out.println(e);}
18.
19.     t2.start();
20.     t3.start();
21. }
22. }
```

Test it Now

```
Output:1
        2
        3
        4
        5
        1
        1
        2
        2
        3
        3
        4
```

```
4
5
5
```

As you can see in the above example, when t1 completes its task then t2 and t3 starts executing.

Example of join(long milliseconds) method

```
1. class TestJoinMethod2 extends Thread{
2.     public void run(){
3.         for(int i=1;i<=5;i++){
4.             try{
5.                 Thread.sleep(500);
6.             }catch(Exception e){System.out.println(e);}
7.             System.out.println(i);
8.         }
9.     }
10. public static void main(String args[]){
11.     TestJoinMethod2 t1=new TestJoinMethod2();
12.     TestJoinMethod2 t2=new TestJoinMethod2();
13.     TestJoinMethod2 t3=new TestJoinMethod2();
14.     t1.start();
15.     try{
16.         t1.join(1500);
17.     }catch(Exception e){System.out.println(e);}
18.
19.     t2.start();
20.     t3.start();
21. }
22. }
```

Test it Now

```
Output:1
        2
        3
        1
        4
        1
        2
        5
        2
        3
```

```
3
4
4
5
5
```

In the above example, when t1 completes its task for 1500 milliseconds (3 times) then t2 and t3 start executing.

getName(), setName(String) and getId() method:

```
public String getName()
```

```
public void setName(String name)
```

```
public long getId()
```

1. **class** TestJoinMethod3 **extends** Thread{
2. **public void** run(){
3. System.out.println("running...");
4. }
5. **public static void** main(String args[]){
6. TestJoinMethod3 t1=**new** TestJoinMethod3();
7. TestJoinMethod3 t2=**new** TestJoinMethod3();
8. System.out.println("Name of t1:"+t1.getName());
9. System.out.println("Name of t2:"+t2.getName());
10. System.out.println("id of t1:"+t1.getId());
- 11.
12. t1.start();
13. t2.start();
- 14.
15. t1.setName("Sonoo Jaiswal");
16. System.out.println("After changing name of t1:"+t1.getName());
17. }
18. }

Test it Now

```
Output:Name of t1:Thread-0
       Name of t2:Thread-1
       id of t1:8
       running...
```

```
After changling name of t1:Sonoo Jaiswal  
running...
```

The currentThread() method:

The currentThread() method returns a reference to the currently executing thread object.

Syntax:

```
public static Thread currentThread()
```

Example of currentThread() method

```
1. class TestJoinMethod4 extends Thread{  
2.     public void run(){  
3.         System.out.println(Thread.currentThread().getName());  
4.     }  
5. }  
6. public static void main(String args[]){  
7.     TestJoinMethod4 t1=new TestJoinMethod4();  
8.     TestJoinMethod4 t2=new TestJoinMethod4();  
9.  
10.    t1.start();  
11.    t2.start();  
12. }  
13. }
```

Test it Now

```
Output:Thread-0  
        Thread-1
```

Naming Thread and Current Thread

Naming Thread

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using setName() method. The syntax of setName() and getName() methods are given below:

1. **public String getName():** is used to return the name of a thread.
2. **public void setName(String name):** is used to change the name of a thread.

Example of naming a thread

```
1. class TestMultiNaming1 extends Thread{
2.     public void run(){
3.         System.out.println("running...");
4.     }
5.     public static void main(String args[]){
6.         TestMultiNaming1 t1=new TestMultiNaming1();
7.         TestMultiNaming1 t2=new TestMultiNaming1();
8.         System.out.println("Name of t1:"+t1.getName());
9.         System.out.println("Name of t2:"+t2.getName());
10.
11.        t1.start();
12.        t2.start();
13.
14.        t1.setName("Sonoo Jaiswal");
15.        System.out.println("After changing name of t1:"+t1.getName());
16.    }
17.}
```

Test it Now

```
Output:Name of t1:Thread-0
        Name of t2:Thread-1
        id of t1:8
        running...
        After changeling name of t1:Sonoo Jaiswal
        running...
```

Current Thread

The `currentThread()` method returns a reference of currently executing thread.

1. **public static** Thread `currentThread()`

Example of `currentThread()` method

```
1. class TestMultiNaming2 extends Thread{
2.     public void run(){
```

```

3. System.out.println(Thread.currentThread().getName());
4. }
5. public static void main(String args[]){
6. TestMultiNaming2 t1=new TestMultiNaming2();
7. TestMultiNaming2 t2=new TestMultiNaming2();
8.
9. t1.start();
10. t2.start();
11. }
12. }

```

Test it Now

```

Output:Thread-0
        Thread-1

```

Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

```

1. class TestMultiPriority1 extends Thread{
2. public void run(){
3. System.out.println("running thread name is:"+Thread.currentThread().getName());
4. System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
5.
6. }

```

```
7. public static void main(String args[]){
8.   TestMultiPriority1 m1=new TestMultiPriority1();
9.   TestMultiPriority1 m2=new TestMultiPriority1();
10.  m1.setPriority(Thread.MIN_PRIORITY);
11.  m2.setPriority(Thread.MAX_PRIORITY);
12.  m1.start();
13.  m2.start();
14.
15. }
16. }
```

Test it Now

```
Output:running thread name is:Thread-0
        running thread priority is:10
        running thread name is:Thread-1
        running thread priority is:1
```

Multithreading in Java

Multithreading in java is a process of executing multiple threads simultaneously.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation etc.

Advantages of Java Multithreading

- 1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.
 - 2) You **can perform many operations together so it saves time**.
 - 3) Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.
-

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

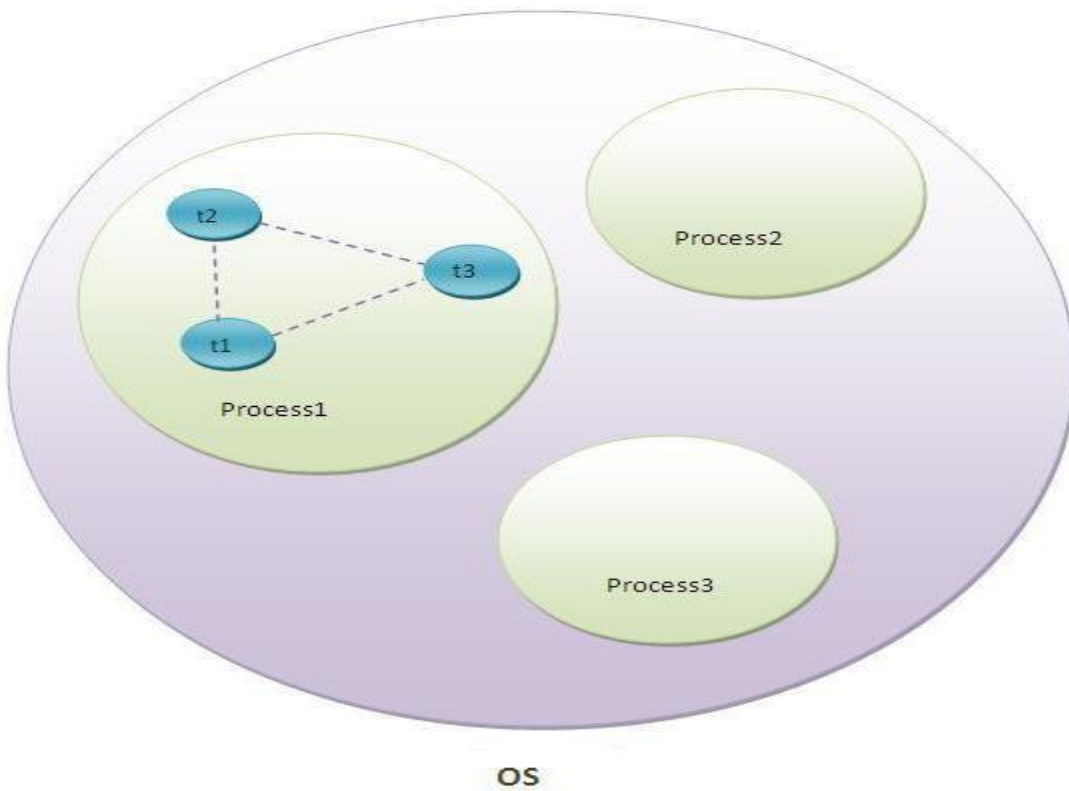
- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.

Note: At least one process is required for each thread.

What is Thread in java

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.



As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

Note: At a time one thread is executed only.

Throws clause in java - Exception handling

As we know that there are two types of exception **checked and unchecked**.

Checked exception (compile time) force you to handle them, if you don't handle them then the program will not compile.

On the other hand unchecked exception (Runtime) doesn't get checked during compilation. **Throws keyword** is used for handling checked exceptions . By using throws we can declare multiple exceptions in one go.

What is the need of having throws keyword when you can handle exception using try-catch?

Well, thats a valid question. We already know we can **handle exceptions** using **try-catch block**.

The throws does the same thing that try-catch does but there are some cases where you would prefer throws over try-catch. For example:

Lets say we have a method `myMethod()` that has statements that can throw either `ArithmeticException` or `NullPointerException`, in this case you can use try-catch as shown below:

```
public void myMethod()  
{  
    try {  
        // Statements that might throw an exception  
    }  
    catch (ArithmeticException e) {  
        // Exception handling statements  
    }  
    catch (NullPointerException e) {  
        // Exception handling statements  
    }  
}
```

But suppose you have several such methods that can cause exceptions, in that case it would be tedious to write these try-catch for each method. The code will become unnecessary long and will be less-readable.

One way to overcome this problem is by using throws like this: declare the exceptions in the method signature using throws and handle the exceptions where you are calling this method by using try-catch.

Another advantage of using this approach is that you will be forced to handle the exception when you call this method, all the exceptions that are declared using throws, must be handled where you are calling this method else you will get compilation error.

```

public void myMethod() throws ArithmeticException, NullPointerException
{
    // Statements that might throw an exception
}

public static void main(String args[]) {
    try {
        myMethod();
    }
    catch (ArithmeticException e) {
        // Exception handling statements
    }
    catch (NullPointerException e) {
        // Exception handling statements
    }
}

```

Example of throws Keyword

In this example the method myMethod() is throwing two **checked exceptions** so we have declared these exceptions in the method signature using **throws** Keyword. If we do not declare these exceptions then the program will throw a compilation error.

```

import java.io.*;
class ThrowExample {
    void myMethod(int num)throws IOException, ClassNotFoundException{
        if(num==1)
            throw new IOException("IOException Occurred");
        else
            throw new ClassNotFoundException("ClassNotFoundException");
    }
}

public class Example1{
    public static void main(String args[]){
        try{
            ThrowExample obj=new ThrowExample();
            obj.myMethod(1);
        }catch(Exception ex){
            System.out.println(ex);
        }
    }
}

```

Output:

```

java.io.IOException: IOException Occurred

```



13.7 Throwing Our Own Exceptions

There may be times when we would like to throw our own exceptions. We can do this by using the keyword **throw** as follows:

```
throw new Throwable_subclass;
```

Examples:

```
throw new ArithmeticException( );  
throw new NumberFormatException( );
```

Program 13.6 demonstrates the use of a user-defined subclass of **Throwable** class. Note that **Exception** is a subclass of **Throwable** and therefore **MyException** is a subclass of **Throwable** class. An object of a class that extends **Throwable** can be thrown and caught.

Program 13.6 *Throwing our own exception*

```
import java.lang.Exception;  
class MyException extends Exception  
{  
    MyException(String message)  
    {  
        super(message);  
    }  
}  
class TestMyException  
{  
    public static void main(Strings args[ ])  
    {  
        int x = 5, y = 1000;  
        try  
        {  
            float z = (float) x / (float) y ;  
            if(z < 0.01)  
            {  
                throw new MyException("Number is too small");  
            }  
        }  
        catch (MyException e)  
        {  
            System.out.println("Caught my exception");  
            System.out.println(e.getMessage( ) );  
        }  
        finally  
        {  
            System.out.println("I am always here");  
        }  
    }  
}
```

Output:

```
Caught my exception  
Number is too small  
I am always here
```

The object `e` which contains the error message "Number is too small" is caught by the **catch** block which then displays the message using the `getMessage()` method.

Note that Program 13.6 also illustrates the use of **finally** block. The last line of output is produced by the **finally** block.



9.7 Wrapper Classes

As pointed out earlier, vectors cannot handle primitive data types like **int**, **float**, **long**, **char**, and **double**. Primitive data types may be converted into object types by using the wrapper classes contained in the **java.lang** package. Table 9.4 shows the simple data types and their corresponding wrapper class types.

Table 9.4 Wrapper Classes for Converting Simple Types

<i>Simple Type</i>	<i>Wrapper Class</i>
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long

Table 9.5 Converting Primitive Numbers to Object Numbers Using Constructor Methods

<i>Constructor Calling</i>	<i>Conversion Action</i>
Integer IntVal = new Integer(i);	Primitive integer to Integer object
Float FloatVal = new Float(f);	Primitive float to Float object
Double DoubleVal = new Double(d);	Primitive double to Double object
Long LongVal = new Long(l);	Primitive long to Long object

Note: *i, f, d and l are primitive data values denoting int, float, double and long data types. They may be constants or variables.*

Table 9.6 Converting Object Numbers to Primitive Numbers Using `typeValue()` method

<i>Method Calling</i>	<i>Conversion Action</i>
int i = IntVal.intValue();	Object to primitive integer
float f = FloatVal.floatValue();	Object to primitive float
long l = LongVal.longValue();	Object to primitive long
double d = DoubleVal.doubleValue();	Object to primitive double

Table 9.7 Converting Numbers to Strings Using to String() Method

<i>Method Calling</i>	<i>Conversion Action</i>
<code>str = Integer.toString(i)</code>	Primitive integer to string
<code>str = Float.toString(f);</code>	Primitive float to string
<code>str = Double.toString(d);</code>	Primitive double to string
<code>str = Long.toString(l);</code>	Primitive long to string

Table 9.8 Converting String Objects to Numeric Objects Using the Static Method ValueOf()

<i>Method Calling</i>	<i>Conversion Action</i>
<code>DoubleVal = Double.Valueof(str);</code>	Converts string to Double object
<code>FloatVal = Float.ValueOf(str);</code>	Converts string to Float object
<code>IntVal = Integer.Valueof(str);</code>	Converts string to Integer object
<code>LongVal = Long.ValueOf(str);</code>	Converts string to Long object

Note: *These numeric objects may be converted to primitive numbers using the typeValue() method as shown in Table 9.6.*

Table 9.9 Converting Numeric Strings to Primitive Numbers Using Parsing Methods

<i>Method Calling</i>	<i>Conversion Action</i>
<code>int i = Integer.parseInt(str);</code>	Converts string to primitive integer
<code>long i = Long.parseLong(str);</code>	Converts string to primitive long

Note: *parseInt() and parseLong() methods throw a NumberFormatException if the value of the str does not represent an integer.*

Wrapper class Example: Primitive to Wrapper

```
1. public class WrapperExample1{
2. public static void main(String args[]){
3. //Converting int into Integer
4. int a=20;
5. Integer i=Integer.valueOf(a);//converting int into Integer
6. Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
7.
8. System.out.println(a+" "+i+" "+j);
9. }}
```

Output:

```
20 20 20
```

Wrapper class Example: Wrapper to Primitive

```
1. public class WrapperExample2{
2. public static void main(String args[]){
3. //Converting Integer to int
4. Integer a=new Integer(3);
5. int i=a.intValue();//converting Integer to int
6. int j=a;//unboxing, now compiler will write a.intValue() internally
7.
8. System.out.println(a+" "+i+" "+j);
9. }}
```

Output:

```
3 3 3
```