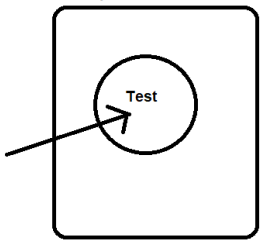
```
class Test{
    // int a;
    // method/function
// How to create function
    access_modifier   ///   return_type   ///  method_name

    public
    static  (optional)
    void/ int/ string/ object
    m1(){

    }
// Two types
    // Instance method
    // static
```

```
export class Test {
    //type scrips
     a number;
```

heep



java

Packages: jar.

encapsulated mechanism releted class / interface/ sub pacakges ---> placing into single module

- **Pre-Define**
  - java.lang; --> heart of Java
  - if we are using class of lang packages class than complier automactically use that class and we no need to import class.
  - java.util;
    java.io;
    java.sql;
    → if you are using class of these packages so you need to import tha classes.
- **User-define**
  - com.netlink
    - variable
    - opertaors
    - if you are using class of user define package so you need to import that class.
- **third-party vender**
  - Connection

Import

If you want to use classes from different userdefine packges/ predefine(expect lang) so you need to import that perticular class or whole packages.

Pre-define

util (collection)

```
import  java.util.ArrayList;

class Test{

   p  s v ma(){

    ArrayList list = new ArrayList();


}
```

user-define

variable

```
public class VariablesDemo {

    private int a;
    private static int b;

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

Operation

```
import com.netlink.variable.VariableDemo;

public class Test {


main(){

 VariableDemo demo =  new VariableDemo();

}
```

Using Import

- **Fully Qualified Name**

```
class Test{


  p s v main(String[ ] args){

  java.util.ArrayList  list  =  new  java.util.ArrayList();

  }

}
```

No need to use import statement;

increase --> code
reduce --> readiablity

- **Short Name**

```
import  java.util.ArrayList;

class Test{


  p s v main(String[ ] args){

   ArrayList  list  =  new  ArrayList();

  }

}
```

less --> code
increase --> readiablity

:Type:

- **Explicit** — you are importing certaine class/ interface

```
import  java.util.ArrayList;
import   java.utill.LinkedList;

class Test{

  main( ) {

     ArrayList al   = new ArrayList();

      LinkedList  ls = new LinkedList();

  }
}
```

- **Implicit** — you are importing whole packages/ class/ interface

```
import java.util.*;

class Test {


    main (){

    ArrayList al = new ArrayList();

    LinkedList  ls = new LinkedList();


  }
}
```

class Modifier:

class Test{

}

----> c , c++ ==> public, private , protected, default ====> Access Specifier
----> reaming ---> access modifier

---> JAVA ---> Every thing Access Modifier

JVM

1. accessibility
2. child class creation possible or not
3.object creation is possible or not

types of classes

Top level classes

public
default
final
abstract *******
strictfp

Inner classes

private
protected
static

```
// Top class
class Test{
  // inner class
  class Test2{
  }
}
```

Note :: IF you are using any other modifier for top level class you will get CE:

if you are not providing any modifier for top level class it will consider it as default

::-- Public clases --:: class / member/ methods

anywhere --> access (within and outside)

```
package pack1;

public class Test{

  pubic void m1(){
    "Hello";
  }

}
```

```
package pack2;
import pack1.Test;
public class Test2{

  p s v main(String[ ] args){

     Test test = new Test();
      test.m1();
  }
}
```

:: default classes :: class/ members/ methods

(Package level access)

```
package pack2;

class Test{
  instance
  public void m1(){

    "Hello...";
  }

}
```

```
package pack2;

class Test2{

  public s v main(S ){

    Test test = new Test();
     test.m1();

  }
}
```

```
package pack1;

class Test3{

  p s v main(S ){

  // why ??
default class can not use outside the package
  }

}
```

::Final Modifier::

final is applicable classes, methods, variables

Final methods;;

final class

```
class Parent {

  public void property(){
    System.out.println("cash + gold + land ");
  }

  public final void marriage( ) {
    System.out.println("shub")
  }

}
```

```
class Child extends Parent{
  @Override
  public void marriage (){
    "abc"
  }
  p s v main(){
    Child c = new Child();
    c.ma();  // public
    c.pro()
  }
```

overriding

```
m1(){
  "hello";
}
```

```
class Test{  // parent

  m1(){
    "hii"
  }
}

// child
class Test2 extends Test{

  p s v main(){

    Test2 t = new Test2();
    t.m1();

  }
}
```

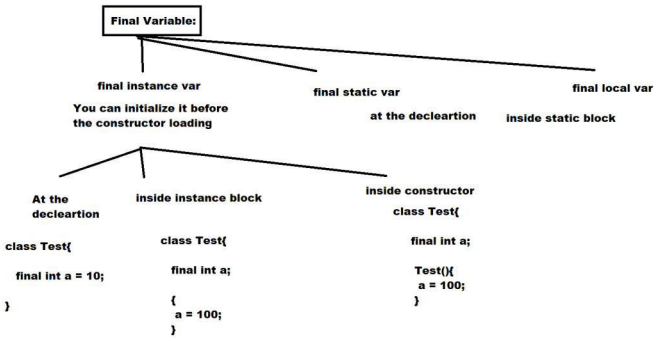
child class object create

Final class

```
final class Parent {

}

class Child extends Parent{

}
```

1. every method present inside a final class is always final by deafult,

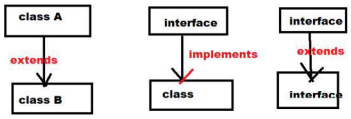
2.but every variable present inside a final class need not be final.


Final Variable:
final instance var
You can initialize it before the constructor loading
final static var
at the decleartion
final local var
inside static block
At the decleartion
class Test{
final int a = 10;
}
inside instance block
class Test{
final int a;
{
a = 100;
}
inside constructor
class Test{
final int a;
Test(){
a = 100;
}

Native::--

Interface ::--- (No Defination) 100% Abstartion

```
interface A {

    //public , abstract --> Method
    public void m1();
}

class B implements A {

   public void m1(){
     "Hii..."
   }

}
```



Interface Method — public, abstract

to available for all implementation class

implementation class is responsible to provide implementation

interface variable — public , static , final

to available for all implementation class

without existing the object we can access this variable

implementation class can access it but can not modify it.

Interface naming conflicts

Method naming conflicts;

Case 1 : interfcaes method --> same

If two interfaces have the same the method with the same signature and the same return type in the implementation class only one implementation is enough.

```
interface Left{
   public void m1();
}
interface Right{
   public void m1();
}

class Test implements Left, Right{

   public void m1(){
     "Hi,,,"
    }
}
```

Variable Nameing

```
interface Left{
   int a = 10; // public , static , final
}
interface Right {
   int a = 20; // public , static, final
}

class Test implements Left, Right{

main(){

   Left.a  // 10
   Right.a   // 20
```

case 2: interface method signature Diff

If two interfaces contained method of diff method signature and if they are implementing in same class than it is compulsory to provide implementation for both interface methods.

```
interface Left {

   public void m1();

}

interface Right{

   public void m1(int a);

}

class Test implements Left, Right{

public void m1(){  "Hi .." );

public void m1(int a){  "Bye.. ");
```

public void m1(){

}

method signature -->

method name ++ Method Parameter

Case :3

interface --> method sig == SAME ,, Return Type != Diff

NOTE:: THERE is no java class that will provide implementation for this methods.

```
interface Left {
   void m1();
}

interface Right{
  int m1();
}

class Test implementation Left, Right{

CE;;;;;;

```

Marker Interface::

An interface that does not have any method. by implementing that interface in class so thatthe that havewill get more ability to perform some action.

1. Serializable
2. Clonable
3. RandomAccess

If we want to share our data to outside network

JVM

client

Student info

req JSON obj

JSON obj

resp

Server

Student {

id;
name;
age;
}

Serializable

implements

```
{
  "id": 1,
  "name":"jay",
  "age":20
}
```

JSON

Adapter Class;

a simple java class that implements in interface only with an empty body.

Adapter Class

```
interface Test{

  m1();
  m2();
   m3();
  m4();
  m5();
}
```

```
class Right implements Test{

   m1(){ }

   m2(){ }

   m3(){ }

   m4(){ }

    m5() { }

}
```

```
class Akshay implements Test{

   m2(){
      "Hii"
   }

    m5(){
      "Hello"
    }
}
```

```
class Test44 extends Right{

   @Override
   public void m1(){
     "Hii"
    }
}
```

OOP's

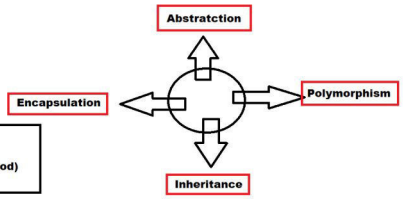
class/ object

oops --> SI --> Most powerfull

Procedural Pro --> without object

OOPs --> writing code with object

Properties/ states (field/variable) | Behaviour (method)




```
class Test{

int a;  // state
```

Data Hiding: Our internal data should not go out directly that outside person can not use our data internally.

Advantage:
1. Security

Recommended :
1. data members - private

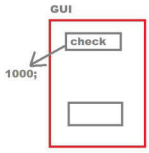
```
class Test{
  int a;  // var  // data hide
}

class Test2 {

  p s v main(Sting[] args){
    Test t = new Test();
      t.a =100;
    }
}
```

```
class Account{

private double balance;

}

class SBI {

   p  s v main(){
     Account a = new Account();
     // a.balance=1000000;   CE:
}
```

Abstraction:

Hide internal implementation and just highlight the set of services is called Abstraction.

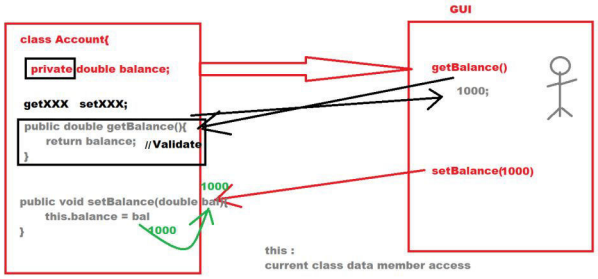
1. abstract class (0-100)
2. interface (100%)

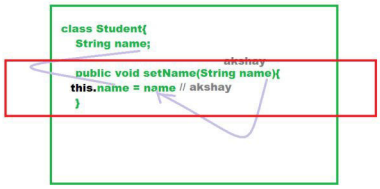


Encapsulation: private DataHiding + Abstraction = Encapsulation

corresponding
Binding --> data + method into a single unit.

DH
Abstarction





Tightly Encapsulated class;

every data members ===> private not req : getXXX and setXXX

```
class A {
    private int a = 10;   // TEC
}
class B extends A{
    int b = 10; // Not
}
class C extends A{
    private int c = 1000; // TEC
}
```

```
class A{
    int a = 10;  // NOT TEC
}
class B extends A{
    private int b = 100;  // Not
}
```

```
class Parent{

   public void getPro(){
     "100Acre"
   }
}
class Child extends Parent{

   public void shop(){
     "Cafe"
   }
}
```

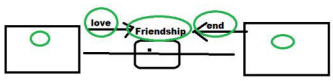
```
class Main{

 public static void main(){

   Child c = new Child();
      c.getPro(); // 100Arc
       c.shop(); // Cafe

   Parent p = new Parent();
      p.getPro();  // 100Arc
      p.shop();   // CE:

  Parent p1 = new Child();
     p1.getPro();
     p1.shop(); // CE:

   Child c2 = new Parent();    // CE:

  }
}
```

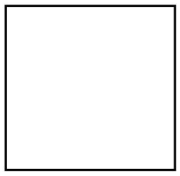
```
class A {
  void m1(){
    "ssss"
  }
  p s v main(){
     A a = new A();

  }
}
```

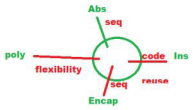
:Polymorphism:

Poly ---> Many
Morphism ---> Forms

Method







type

Compile time
static / early binding

Run-time/ dynamic / late binding

OverLoading

method hiding

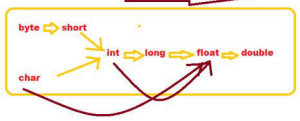
Overriding

Defi:
two methods -->

name --> same
agruments --> different ==> method signature different
(Method Signature ==> method name + args)
class --> same

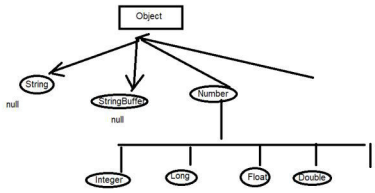
method overloading

case 1:

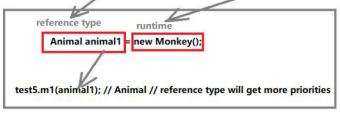


==> Exact match will get higher priority (No automatic promotion)
==> Exact match is not there (automatic promotion)

```
class A {

   m1(){  1 }
   m1(int a){  2 }

p s v main(String[] agrs){

   A a = new A();
   a.m1();  // 1
   a.m1(2);  // 2
}
```



In the overloading method resolution is always based on reference type and runtime objects won't play any role in method overloading.



overriding --> inheritance

In the overriding method, the resolution is always taken by JVM based on the runtime object.

RUN TIME POLYMORPHISM

Reference type

```
Parent parent1 = new Child();

parent1.property(); // "Cash + Gold"
parent1.marry();    // Avi..
```

Rule:

1. overriding

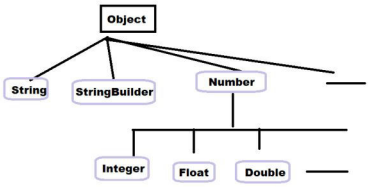
method signature -- SAME
same -- name
same -- argument
class -- different

2. overridden --> Parent class method
overriding --> Child class override method

```
class Parent {        parent
   public Object m1(){
      return null;
   }
}

class Child extends Parent{

// override--   child
   public String m1(){  // overriding method
      return "String";
   }
}
```

3. up to 1.4v override same return type

4. from 1.5v you can use a co-variant return type:
return type ---> may be different




```
class A {
   public A m1(){
      return new A();
   }
}

class B extends A {
   public B m1(){
      return new B();
   }
}
```

this : ---> current class

variable this.va
method this.m1()
constructor

parameter non-parameter

super :-
refer --> super class object/instance

variable
method
constructor

keyword	constructor calls
this/super anywhere in instance method	this()/ super() first line of constructor

Constructor:

```
Class Test{
  int a;
  main(){
    Test test = new Test();
    test.a = 100;
  }
}
```

```
class Student{
  private int id;
  public int getId(){
    return id;
  }
  p v setId(int id){
    this.id = id;
  }
}
class Main{
  main(){
    Student s1 = new Student();
    s1.setId(100);
    sop(s1.getId());  // 100
  }
}
```

```
class Employee{
  int id; 1
  String name; ABC
  // Constructor
  Employee(int id, String name){
    this.id = id;
    this.name = name;
  }
  p s v main(){
    Employee e1= new Employee(); // CE
    Employee e2 = new Emplyee(1, "ABC");
    Employee e3 =  new Employee(2,"XYZ");
```

e2 → id // 1, String // ABC

e3 → id 2, name XYZ

```
class Test{
  Test(){
    super();
  }
}
class Main{
  main(){
    Test t = new Test(); // default(JVM) // no-agrs // if there is no constructor
```

JVM --> super();

Programmer → JVM

```
class Test{ }
```
→
```
class Test{
  Test(){
    //super();
  }
}
```

```
public class Test { }
```
→
```
public Test{
  public Test(){
    //super();
  }
}
```

```
class Test{
  void Test() { } //method
}
```

you can create a method name is same as class name but this not a programming practice. JVM consider this method as a Normal method

```
class Test {
  Test(){
    // super();
  }
  void Test(){
    //some output
  }
```

```
class Test{
  Test(int i) { }
}
```

if you are creating any constructor, then JVM is not going to provide any default constructor

```
class Test {
  Test(int i){
    super();
  }
}
```

```
class Test{
  Test(int i){
    super();
  }
}
```

if you are creating any constructor, then JVM is not going to provide any default constructor

```
class Test{
  Test(int i){
    super();
  }
}
```

Test t = new Test(); // CE
Test t = new Test(10); // CE

```
class Test {
  Test(int i ){
    this();
  }
  Test(){
  }
}
```
→
```
class Test {
  Test(int i){
    this();
  }
  Test(){  // user created
    super();
  }
}
```

```
clas Test{
  Test(){  //non-para
  }
  Test(int i ){  // para
  }
}
```