JAVA (Object Oriented Programming)

JAVA Source File Structure

-> There can be only One Public class.

-> There can be any number of classes.

-> If There is a Public class, The name of programe should be Same as Public class name.

-> If There are no public class, Present, We can name programe whatever we want.

-> Every Class Contains A main method();

-> whenever a .class is executed, The Corresponding main method will execute.

-> If there is no main method in a class, The error occurs.

Example:-

Test.java

```
public class Test {
        public static void main (String[], args) {
                System.out.println ("Main class");
                }
        }

Class A {
        public static void main (String[], args){
                System.out.println ("A class");
        }   }
```

Import Statement Introduction.

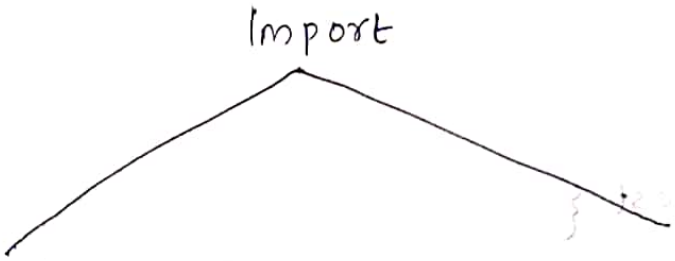
-> Import statement improves readability, Reduces length of code by using short names instead of fully qualified names.-(java.util.ArrayList)

example :-

Test.java

```
import java.util.ArrayList;
public class Test {
        public static void main (String[] args)
            {
              ArrayList l = new ArrayList();
            }
}
```

-> Two Types of Import.



Explicit Import

import java.util.ArrayList;

specific packages

Good to use.

Implicit Import

import java.util.*;

whole packages.

Not good practice.

Readability reduces

→ Import State ment not needed for some

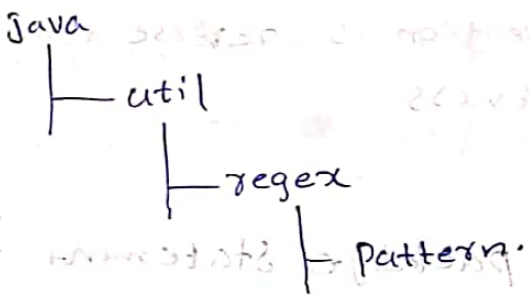
① java. lang
   └ common terms like String, Thread ... etc.

② Default Package.
   └ Correct working directry

→ whenever we're importing a package all classes and Interfaces present in that package by default available, but not sub package class.

If we want to use sub package, The import statement Should be upto the sub package level.

example:-



① import java.* ; — error,

② import java.util.* ; — error.

③ import java.util.regex.* ; — works fine.

④ Import java.util.regex.Pattern ; — works fine.

Package

-> Group of related things. (General)

-> Group of related Classes & Interfaces.

-> Encapsulation mechanism to group related Classes and Interfaces into a single group.

① We can resolve Naming Conflict
② Modularity Improved
③ Maintainability.
④ Security.

example:-

```
package com.durgasoft.oci;
```

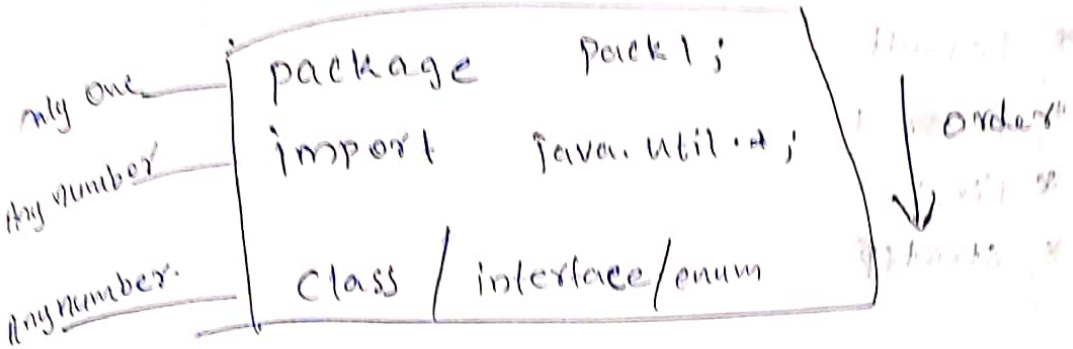
Naming Convention is reverse domain name. for uniqueness

-> ~~Only One~~ Atmost One package statement is allowed in a programme.

```
import java.util.*;
Package Pack1;
public Class {
}
```

- Error.

→ Always Put Package statement on top of programme. (should be first statement.



Class level modifiers : Public & Default

→ Modifiers Describes behaviour of the class

→ Public

class is accessible anywhere. within or outside the package.

→ Default

Accessible only within the package.

→ Abstract

object creation is not possible.
Cannot be instanciated.

→ final

child class creation is not possible.

~~→ strictfp~~

strict floating point

abstract void m1();

→ For Toplevel classes, Applicable Modifiers Are.

* Public
* Default
* Abstract
* Final.
* strictfp

→ For Inner classes. (class inside class)

* Private
* Protected
* Static.

Toplevel Class Test {

Inner class Inner {

}

}

```
package pack1;
public class A
{

}
```

```
Package pack2;
import pack1.A;
Public Class B
{
Public static void main (String[])
{
A a = new A();
}
}
```

No Error.

By Default its Default

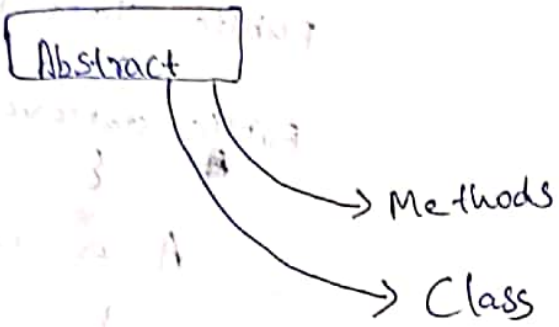
```
packager Pack1;

class A
{

}
```

```
package pack2;
import pack1.A;
public class B
{
psvm (String[] args)
{
A a = new A();
}
}
```

Error.

Can't access Default class out side package.

Abstract Method



Abstract Method

-> It has only Declaration No Definition/ implimentation.

-> This method end with semicolumn (;).

-> Abstract methods cannot have body. * The child classes will provide implimentation

example:-

```
abstract Class loan {

        public abstract double getInterest();

        }
```

1. public abstract void m1() {} ✗
2. public void m1(); ✗ Not valid.
3. public abstract void m1(); ✓
4. public void m1() {} ✓ Valid

-> If Class contains a abstract method The class should be abstract class.

Abstract Class (Partially Implimented class)

-> If the implimentation of class is not complete. The class should be declared as abstract.

-> No one is allowed to create an object of abstract class

-> No one is allowed to call these methods directly.

Abstract Class V/S Abstract Method

-> If a class contains atleast one abstract method. Compulsory The class should be declared as abstract.

-> Even though there are no abstract methods. The Class can be declared as abstract.

-> The implimentation of abstract class methods in the child class.

```
abstract class Test {

    Public abstract void m1();
    public abstract void m2();
        }
abstract class Subtest extends Test {

        Public vod m1(){
                }
        }

    class Set extends Subtest {
            public void m2(){
                }
        }
```

class
int v
char c

Example :-

```
abstract class Vehicle {

    public abstract int getNoOfWheels();
        }

class Bus extends Vehicle {

        public int getNoOfWheels(){
            return 6;
            }
        }
```

Eta

~~Vehicle v = new Bus();~~

~~Bus v = new Vehicle();~~

```
Class Auto extends Vehicle {
    public int getNoOfWheels(){
        return 3;
    }
}

Class Test {
public static void main (String [] args)
    {
    Bus obj = new Bus();
    Auto obj1 = new Auto();
    SOP (obj.getNoOfWheels());
    SOP (obj1.getNoofWheels());
    }
}
```

5 f%= 2, 3 5 3 = 0

f%2 = j = 1 j+=2 f = 4 3%0 = 1

j = 2 j+1=3 j+3=5 3 = 0 fla

i 2 3 6 9 8 9 10 11 12 10 0

j j+1 j+2 j+3

5 4 5 6 9 8 9 10 11 12

1 2 5 6 7 8 9 10 11 12

```
for (i=0; i<arr.Leng, i++)
{
    flag=0;
    for(j=1; j<=arr[i]; j++)
    {
    if (arr[i] %j==0)
    {
    flag++;
    }
If (flag==2)
```

Member Modifiers : public and default

Public :-

```
package pack1;

public class A
    {
    public void m1()
        {
        SOP("A class method");
        }
    }
```

```
Package pack2;
import pack1.A;

Public class B
    {

    PSVM
        {
        A a = new A();
        a.m1();
        }
    }
```

Error occurs.

Even if the method is public, The class is not public:

⇒ The Class containing method should be public for public member to be accessible.

∴
```
package Pack1 ;

public class A
{
  public void m1()
  {

  }
}
```

⇒ The Public member can be accessed anywhere.

Default :-

-> Access only in the respective package.

-> Also called package level Access.

Private :-

-> Access Only in the same class.

-> Also called class level Access

-> Highly Recomended for variables.

```
class Bank {
  Private balance;
  Void getbalance()
  {
  If-C
  bal = resul
  retur
  }
```

* Recomended modifier for method is

```
public:
class mini extn
Min
```

Protected :-

→ Access in own package and child classes only.

```
Class Test {
    protected void m1() {
        SOP("A class method");
    }
}
class A extends Test {
    psvm (String [] args) {
        Test a = new Test();
        a.m1();
        A b = new A();
        b.m1();
        Test a1 = new A();
        a1.m1();
    }
}
```

A a2 = new Test(); ✗ error

```
package pack1;
public class A {
    protected void m1(){
        SOP("A class Protected Method)..
        }
    }
```

```
Package pack2;
Import pack1.A;
public class B extends A
    {
    PSVM (String[] args)
      {
      A a = new A();  ✗
       a.m1();
      B b = new B();  ✓
       b.m1();
      A a1 = new B();
       a1.m1(); ✗
      }
    }
```

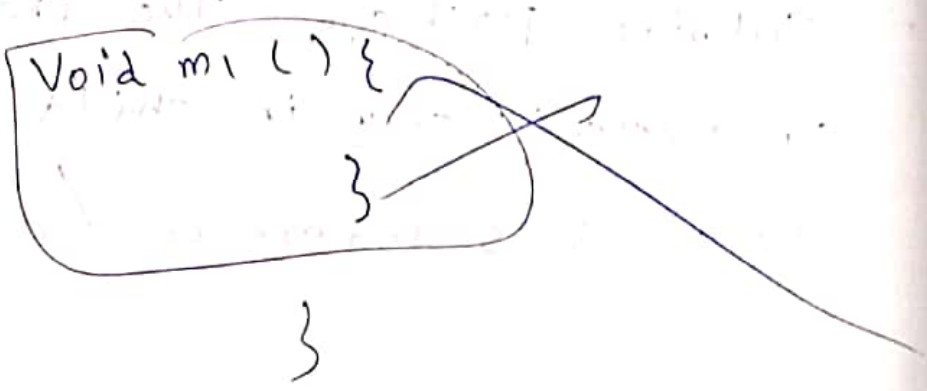
-> Outside package The protected members can accessed only in child class with. Child class reference / child class object.

Summary of Public, Protected, Default, Private

Visibility	Public	Protected	default	Private
within Same class	✓	✓	✓	✓
From child class of same package	✓	✓	✓	✗
From Non-child Class of same Package	✓	✓	✓	✗
From child class of out side package	✓	✓ should use child reference	✗	✗
Non-child class of out side package	✓	✗	✗	✗

Private < default < Protected < Public

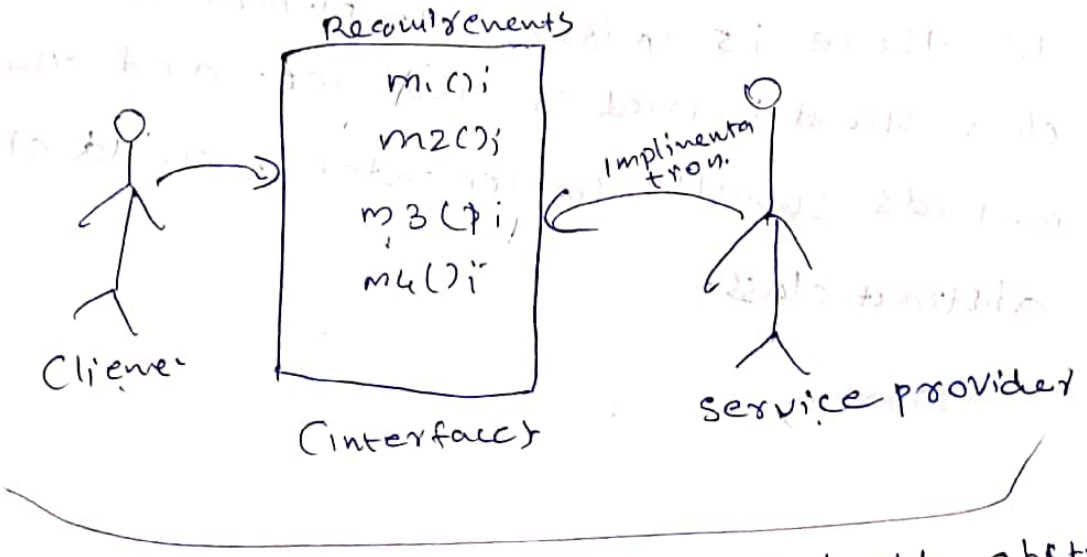
```
Interface Sheberer {

    void m2();

    Void m1() {
    }
}
```



Interface

-> Any Service Requirement Specification (SRS)

-> Any contract b/w Client & Service provider itself is considered as interface.



-> Every method in interface is by default abstract and public.

Hence. no definition. & 100% abstraction.

Interface Declaration & Implementation

```
interface Test
{
  public void m1();
  public void m2();
}
```

```
abstract class Sub implements Test
{
  public void m1()
  {
  }
}
class set extends Sub
{
  Public void m2();
  {
  }
}
```

→ The Access level should be Same (Public) in implimentation of interface methods.

→ The implamented class should contain the implimentation of all methods in interface.

→ If there is missing implimentation, the class should declared as abstract. and the remaining methods should be Implemented in child class of abstract class.

DATA HIDING [OOPS]

⇒ By Declaring Variable as private we can get data biding

⇒ So the private data can be accessed using public method.

⇒ In public method, some validations are carried out like if ( ) { } and if condition satisfies. The private data is returned.

example:-

```
class Account
{
private double balance;

public double getbalance ()
    {
    // validation
     if (valid)
       {
       return balance;
       }
    }

}
```

Abstraction

→ Hiding Internal Implementation and highlighting set of services offered

Advantages:-

① Security

② Enhancement

③ Maintainability

④ Modularity

Encapsulation

→ Process of Grouping Datamembers under corresponding methods into a single unit.

→ Every Java class is an example for encapsulation.

example:-

```
Class Student
    {
      String name;
      int age;
      ...
      read () { }
      write () { }
    }
```

→ Encapsulation = Data hiding + Abstraction

→ If any component follows Data hiding & Abstraction. then it is encapsulated.

```
class Account
    {
     private double balance;

public void getBalance ()
        {
          // validation
          return balance;
        }
public void setBalance (double. amount)
        {
          // Validation.
          this.balance = this.balance + amount;
        }
```

getter & setter methods

Tightly Encapsulation

→ If the all variables are declared as private. then it is tightly encapsulated.

→ If the, class is a child or subclass. The parent class also should be Tightly encapsulated.

Inheritance

1. IS-A Relation Ship
2. Code Reusability.
3. extends key word

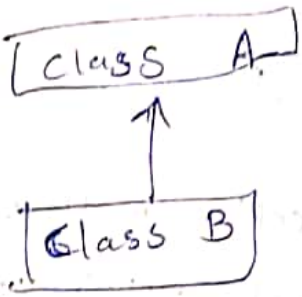
example:-

```
class P {
    public void m1(){
        SOP("Parent");
    }
}
class C extends P{
    publicvoid m2(){
        SOP("child");
    }
}
class Test{
    psvm (string[].args)
    {
    P p = new P();
    C c = new C();
    p.m1();
    c.m1();
    c.m2();
    }
}
```

p.m2(); ✗

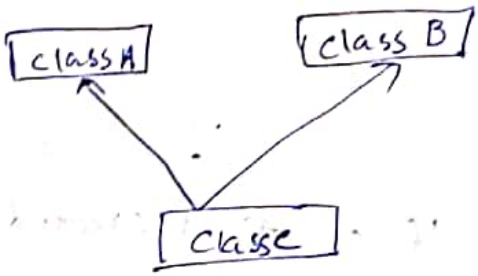
Types Of Inheritances

① Single Inheritance.



Class B extends A

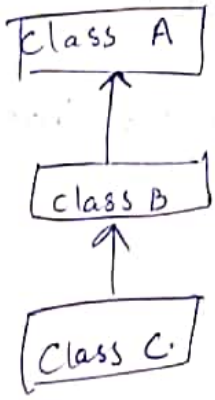
② Multiple Inheritance.



class C extends A, B

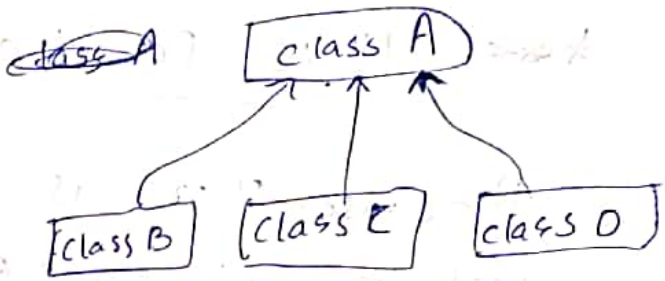
Nb:-
Not Supported in java

③ Multi level Inheritance




```
class A{
    }
class B extends A
    {
    }
class C extends B
    {
    }
```

④ Hierarchical Inheritance.



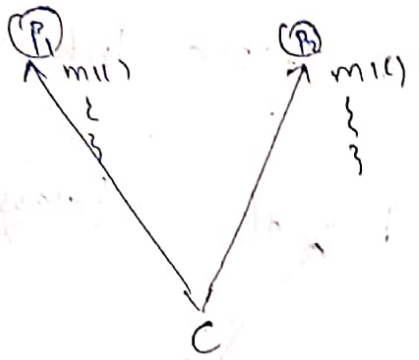

```
class A {
    }
class B extends A
    {
    }
class C extends A
    {
    }
class D extends A
    {
    }
```

⑤ Hybrid Inheritance.

-> Using multiple types of Inheritances at same time.

Nb:- Not Supported in Java.

Multiple Inheritance.



C extends P_1, P_2

If an object created for C on calling method m_1 There will be a conflict in which m1() is called either P_1 or P_2. Thus resulting an error.

Also known as
-> Diamond Access Problem

-> Ambiguity Problem.

★★★ It is Possible in Interface.

Because, There is no definition in the interface methods.

Cyclic Inheritance.

-> Not supported in java.

```
class A extends A
{
}
```

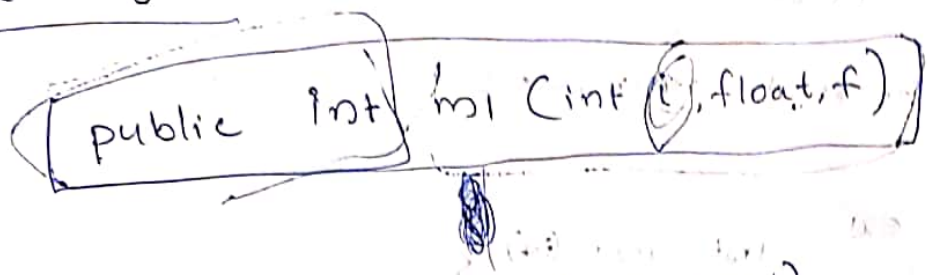
```
class A extends B
{
}

class B extends A
```

-> Not really requiered.

~~Cyclic Inheritance.~~

~~Not S~~

Method Signature



method signature => m1 (int, float).

-> method signature is the method name followed by the argument data types. (order also important).

-> In java return type is not important

-> In a class Two methods with same signature is not allowed

```
class Test {
    public void m1 (int i)
    {
    }
    public int m1(int i)
    {
        return 10;
    }
    Test t = new Test();
    t.m1();
}
```

compile time error: Two methods with same signature

Poly morphism

Method Overloading

-> If bothe methods having same name, but different arguments.

ex:-

```
int m1 (int) {
        }

int m1 (float) {
        }
```

-> Also called Compile-time poly morphism.

-> Static Poly morphism

-> Early Binding.

Case Study -1 (Automatic Promotion)

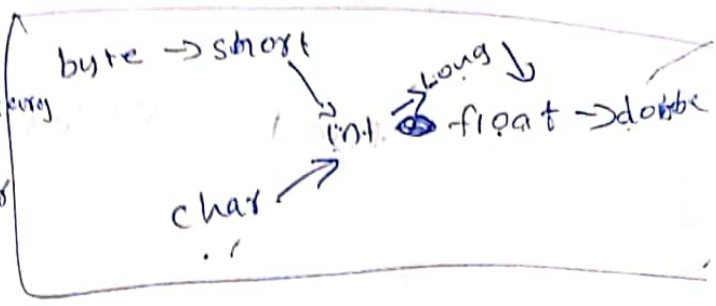
```
class Test
{
    public void m1(int i)
    {
        sopln ("int arg");
    }
    public void m1(float f)
    {
        sopln ("float arg");
    }
    psvm (String[] args)
    { Test t = new Test();
      t.m1(10); // int arg
      t.m1(10.5f); // float arg
```

t.m1('a'); // int arg.

t.m1(10L); // float arg

t.m1(10.5); // Error

}



→ Here the the character 'a' is promoted to the int method.

→ This is called automatic promotion in overloading.

→ The compiler will check for the nex higher type in listed arguments.

→ If the type exeeds all listed types then compiler gives error

Case Study-2

```
class Test
{
    public void m1(int i, float f)
    {
        SOPln("int-float");
    }
    public void m1(float f, int i)
    {
        SOPln("float-int");
    }
    PSVM(String[] args)
    {
        Test t = new Test();
        t.m1(10, 10.5f); // int-float
        t.m2(10.5f, 10); // float-int
```

t.m1(10,10); // compiler Error
! reference to m1 is ambiguous
Both method s match.
(Auto promotion)

t.m1(10.5f,10.5f); // compile error
: No suitable method.

Case Study - 3

```
cass Animal { }
clas monkey extends Animal {}
class Test
{
    public void m1(Animal a)
    { sopln("Animal ver");
    }
    public void m1(Monkey m)
    { sopln("Monkey version");
    }
    psvm(String[] args)
    { Test t = new Test();
      Animal a = new Animal();
      t.m1(a); //Animal vers.
      Monkey m = new Monkey();
      t.m1(m); //Monkey versio
      Animal a1 = new Monkey();
      t.m1(a1); //Animal vers
    }
```

Method Overriding

m1(int)
m1(float)

```
class P {
    public void property()
    {
        sopln("cash + gold");
    }
    public void marry()
    {
        sopln("Appkammma");
    }
}

class C extends P
{
    public void marry()
    {
        sopln("katrina");
    }
}

class Test {
    psvm(String[]args).
    {
        P p = new P();
        p.marry(); -> parent method.
        C c = new C();
        c.marry(); -> child method.
        P p1 = new C();
        p1.marry(); -> child method.
    }
}
```

Overriding method.

P p = new C()
p.marry();

-> Runtime polymorphism.

-> Dynamic

-> Late binding

Rules for Overriding

```
class P
{
    public void m1(int i)
    {
    }
}
class C extends P
{
    public void m1(int i)
    {
    }
}
```

#1. Method Signature Should be Same.

#2. Same Return type. until Version 1.4.

```
class P
{
    private void m1(int)
    {
    }
}
class C extends P
{
    private void m1(float)
    {
    }
}
```

#3. Overriding Concept is not applicable for private methods.

* The private methods cannot be accessed outside class.

// Valid, The Parent class class m1() & child class m1() are independent methods since private

```
class P
{
    public final void m1()
    {
    }
}
class C extends P
{
    public void m1()
    {
    }
}
```

// compile Error

#4. The final methods cannot over ride.

* Final implies that there is no other implementation possible.

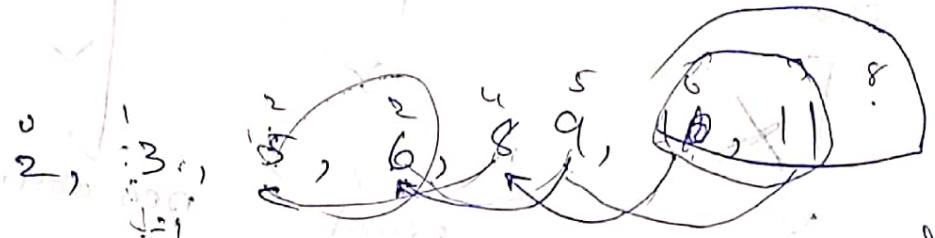
Parent method:

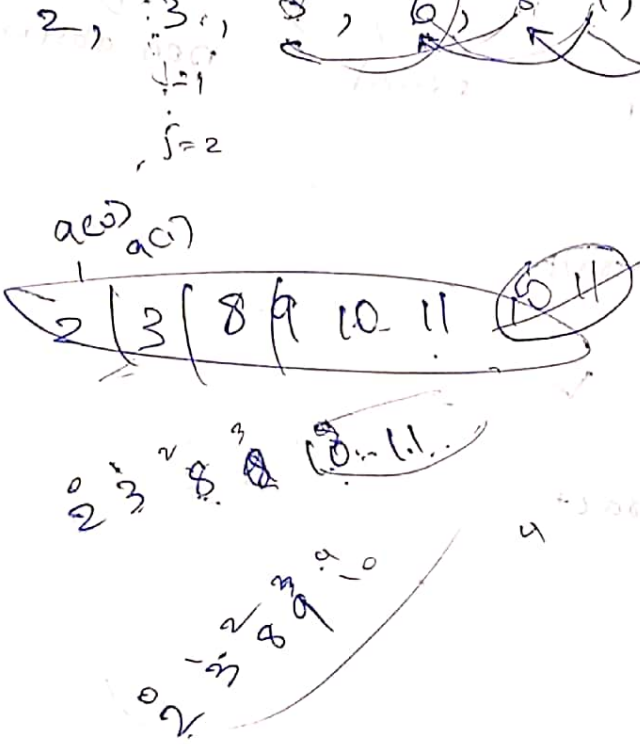
- final → non-final / final
- non-final → final
- abstract → non-abstract

Child method:

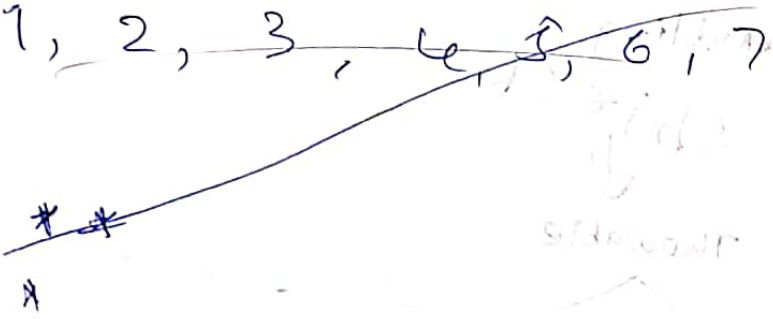
- non-abstract → abstract

```
for( i=0; i< limit ;i++)
{ if( i < lim-1) {
    if( a[i] %2 != 0)
    {
    for( j= i; j< limit ;j++)
    {
        a[j+1] = a[j+3]
    }
    limit = limit-2;
    }
}
}
```





1, 2, 3, 4, 5, 6, 7, 8, 9



Private < default < Protected < Public.

Parent method : Public — Protected

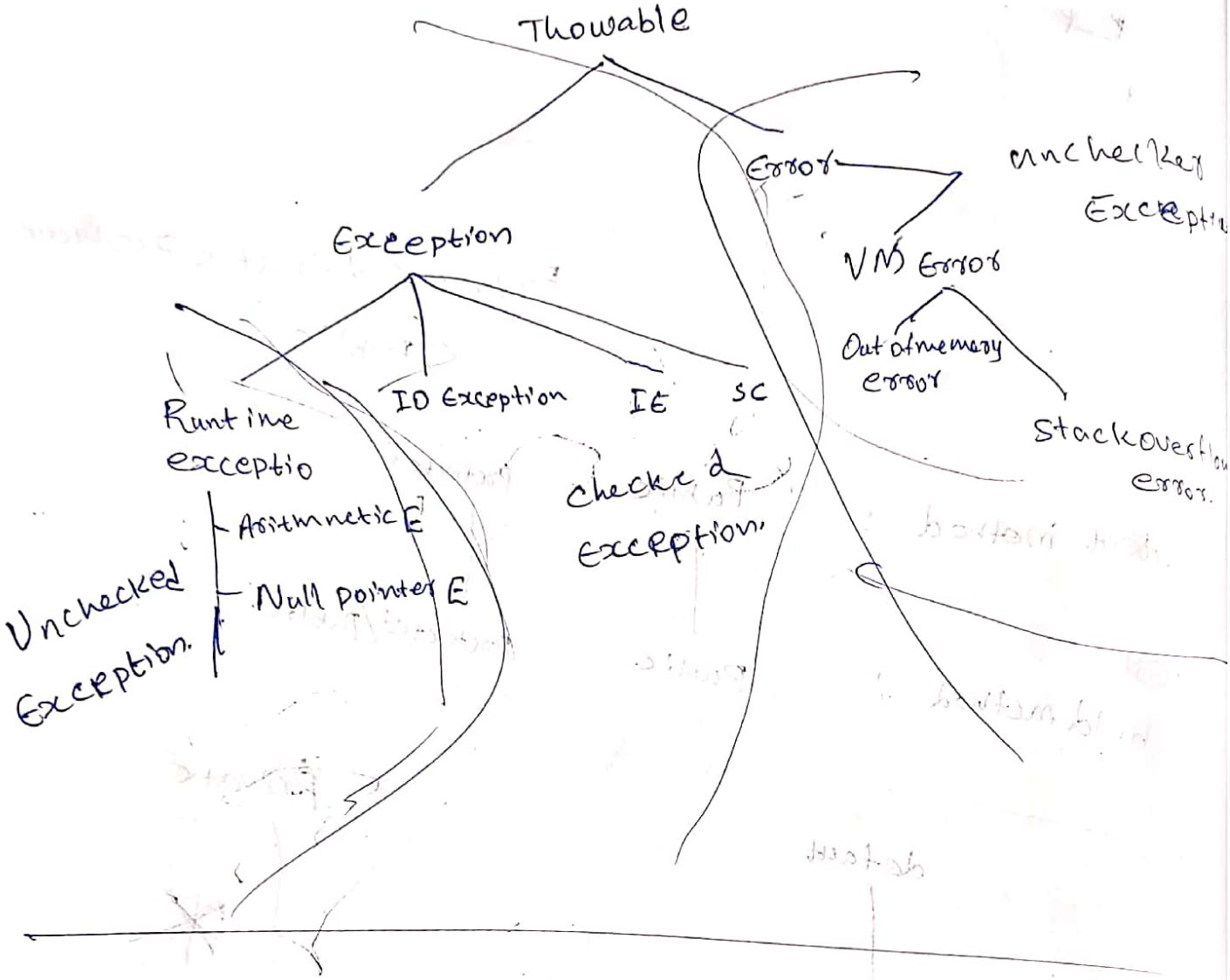
Child method : Public — Protected/Public

Private

default

default/protected/Public

Exception Handling



-> If child class method throws a checked Exceptio
Compulsory ~~to same~~ parant class should throw
same checked exception or its parent

=> For unchecked exception, No rule.

finalize -> Dislinks.
finally -> Like
final ->

① P: public void m1() throws Exception,

C: public void m1() { }

② P: PV m1() { }

C: PV m1() throws exception

③ P: PV m1() { } throws Exception

C: PV m1() { throws IO exception

⇒ Parent method should throw always parent exception

Overriding with static methods

```
class P
  {
    public static void m1() {
                }
  }
class C extends P
    {
    public void m1()
          {
          }
    }
```

```
class P
  {
  Public void m1()
    {
    }
  }
class C extends P
  {
  public static void m1()
      {
      }
  }
```