Declarations & Access Modifiers

1. Java Source File Structure
2. Class Level Modifiers
3. Member Level Modifiers
4. Interfaces

Java Source File Structure

→ A java program can contain any number of classes, but atmost 1 class can be declared as public. If there is a public class, then name of the program & name of the public class must be matched otherwise we will get C.E.

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

case 1 : No public class, we can use any name, no restriction.

A.java ✓
B.java ✓
C.java ✓
Durga.java ✓

```
public  class A { }
        class B { }
        class C { }
```

case 2 : If class B is public, then name of program should be B.java, or we will get C.E.

C.E. class B is public, should be declared in a file named B.java

```
class A
public class B
public class C
```

case 3 : If class B and C declared as public & Name of the program is B.java then we will get C.E.

C.E. class C is public, should be declared in a file named C.java

→

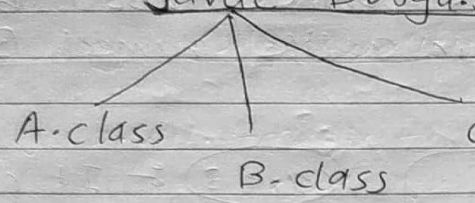
```
class A
{
    p s v main(String[] args)
    { S.O.P("A class Main"); }
}
class B
{
    p s v main (String[] args)
    { s.o.p ("B class Main"); }
}
class C
{
}
```

Durga.java

Scanned with CamScanner

Date / /
Page

* javac Durga.java



* java A ↵ | java B ↵

o/p : A class Main | o/p : B class Main

java C ↵ | java Durga ↵

R.E. | R.E.

NoSuchMethodError : main | NoClassDefFoundError : Durga

(so, Name of the Java program & class containing main does not have any relation)

Conclusions :

① Whenever we compile a java program, for every class present in program a separate .class file will be generated.

② We can compile a java program (java source file), But we can run a java .class.

③ Whenever we are executing a java class, the corresponding class main method will be executed, if class doesn't contain main method, we will get R.E. NoSuchMethodError : main.

④ If the corresponding .class not available then we will get R.E. NoClassDefFoundError

⑤ It is Not Recommended, To declare Multiple classes in a single source file,

★ It is Highly Recommended To declare only 1 class per source file and name of program, we have to keep same as class name. Main Advantage of this approach is Readability & Maintainability of the code is improved.

→ import statement

```
class Test
{
    P S V main(String[] args)
    {
        ArrayList al = new ArrayList();
    }
}
```

C.E. Cannot find symbol
Symbol : class ArrayList
location : class Test

→ We can solve this problem by using Fully Qualified Name

```
java.util.ArrayList al =
        new java.util.ArrayList();
```

→ The problem with the usage of fully qualified name everytime is, it increases the length of the code & Reduces Readability.

→ We can solve this problem using import statement., wherever we use this, it is not require to use fully qualified name everytime, we can use short name directly

```
import java.util.ArrayList;
ArrayList al = new ArrayList();
```
✓

Hence, import statement Act as "typing shortcut"

Case 1 : Types of import statements

(java.util.ArrayList) [Explicit class Import]

(java.util.*) [Implicit class Import]

Readability ✓(↑)
Highly Recommanded

Case 2 : Which import statements are meaningful (?)

a) import java.util.ArrayList; ✓
b) import java.util.ArrayList.*; ✗
c) import java.util.*; ✓
d) import java.util; ✗

(✓ Meaningful, ✗ Not)

Case 3 : class myObject extends java.rmi.UnicastRemoteObject
{
}

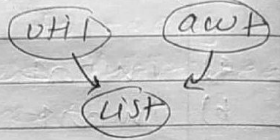
compiler fine ✓, Even though we don't write import stmt, but we use fully qualified path

so, import & FQN are Alternatives

Case 4 :
```
import java.util.*;
import java.sql.*;
class Test
{
    p s v main(String[] args)
    {
        Date d = new Date();
    }
}
```
✗

C.E. reference to Date is ambiguous

Even in the case of List also, we will get same ambiguity problem also, as it available in both pckg o util & awt.



Case 5 :- While resolving class name, Compiler will always give precedence in following order:

1) Explicit class import
2) Classes present in current working directory (CWD) (default package)
3) Implicit class import

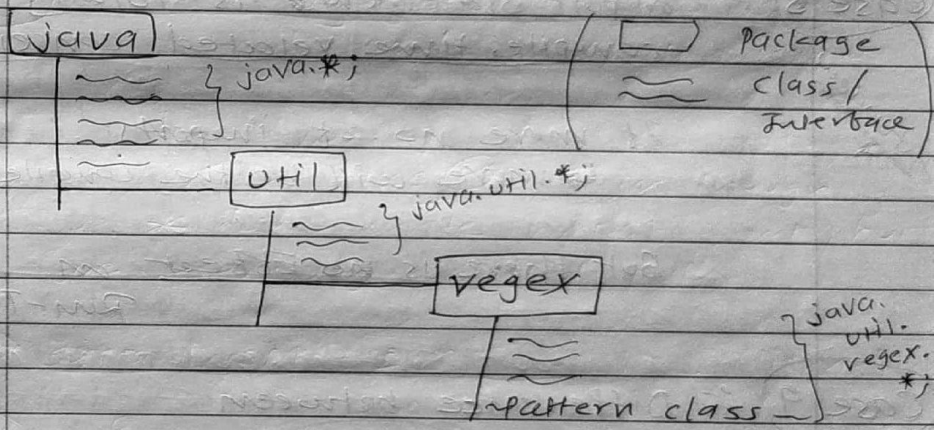
```
{ import java.util.Date;
  import java.sql.*;
  class Test
  {
     p s v main(String[] args)
     {
        Date d = new Date();
        S.O.P(d.getClass().getName());
     }
  }
```

o/p :- java.util.Date;

(Here, util.Date is considered)

Case 6 :- Whenever we are importing a java package, all classes & Interfaces present in that package by default available, But not subpackage classes

If we want to use subpackage class, compulsory we should write import statements until subpackage level



To use Pattern class which import statement is required (?)

a) import java.*;
b) import java.util.*;
c) import java.util.regex.*; ✓
d) import Not Required

Case 7 :- All classes / Interfaces present in the following packages are by default available to every java program, so we don't need to use import statement.

① java.lang
② default package (CWD)
(current Working directory)

Case 8 :- Import statements is totally compile time related concept

Interview ★

If more no. of imports
↳ more will be the compile Time.

But there is no Effect on Run-Time

Case 9 :- Difference between c language #include & Java language import statements.

Interview ★

→ In the case of #include, all i/p, o/p header files will be loaded at the begining only (At Translation time)
∴ static include

→ But In the case of java import stmt, No .class file will be loaded at the begining, whenever we are using a particular class, then only corresponding .class file will be loaded.
∴ Load on Demand / Load on Fly / Dynamic Include

→ java 1.5 V (NEW Features)

(1) For-each loop
(2) Var-arg Methods
(3) AutoBoxing & AutoUnboxing
(4) Generics
(5) Co-variant return types
(6) Queue
(7) Annotations
(8) enum
(9) static import

→ Static import

According to SUN, this feature reduces the length of the code & improves readability,
But According to World wide programming experts, usage of this feature, creates confusion & reduces readability, Hence if there

is no specific requirement, it is not recommanded.

without static import	with static import

class Test
{
psvm(s-g
{
s.o.p(Math.sqrt(4));
s.o.p(Math.max(10,20));
s.o.p(Math.random());
}
}
	import static
	java.lang.Math.sqrt;
	import static
	java.lang.Math.*;
	class Test
	{
	psvm(s-)
	{
	s.o.p(sqrt(4))
	s.o.p(max(10,20))
	s.o.p(random())
	}
	}

→ Explain about `System.out.println();` (Interview ★)

```
class Test
{
    static String s = "java";
}
```

`Test.s.length()`

- Test is class name
- s is a static variable present in a Test class of type java.lang.String
- length() is a method present in string class.

```
class System
{
    static PrintStream out;
}
```

`System.out.println()`

- System is a class present in java.lang pkg
- out is a static variable present in System class of type PrintStream
- println() is a method present in PrintStream class

Hence, we can access by using class-name system.

BUT

Whenever we are using static import it is not require to use class name.

```
import static java.lang.System.out;
class Test
{
    p s v main(String[] args)
    {
        out.println("Hello");
        out.println("Hii");
    }
}
```

o/p & Hello
Hii

Date / /
Page

→ import static java.lang.Integer.*;
import static java.lang.Byte.*;

public class Test
{
psvm(stm~)
{
s.o.p(MAX_VALUE); (X)
}
}

C.E. reference to MAX_VALUE is ambiguous

→ while resolving the static members, the compiler will always consider the precedence / priority in the following order.

(1) Current class static member
(2) Explicit static import
(3) Implicit static import

① → import static java.lang.Integer.MAX_VALUE;
② → import static java.lang.Byte.*;

public class Test
③ { → static int MAX_VALUE = 999;
psvm(stm~)
{

Date / /
Page

s.o.p(MAX_VALUE);
}
}

O/P : 999

→ If we comment Line ③
O/P : 2147483647 (Integer max)

→ If we comment Line ① & Line ③
O/P : 127 (Byte max)

→ Conclusion on import

Normal import (To import classes/Interfaces)

(1) Explicit import

import packageName.className;
e.g.
import java.util.ArrayList;

(2) Implicit import

import packageName.*;
e.g.
import java.util.*;

Date / /
Page

Static import (To import static members of class/Interfaces)

(1) Explicit static import

import static packageName.className.staticMember;

e.g.

import static java.lang.Math.sqrt;
import static java.lang.System.out;

(2) Implicit static import

import static packageName.className.*;

e.g.

import static java.lang.Math.*;

→ Which import stmts are valid?

import java.lang.Math.*; ✗
import static java.lang.Math.*; ✓

import java.lang.Math.sqrt; ✗
import static java.lang.Math.sqrt(); ✗

import java.lang.Math.sqrt.*; ✗
import static java.lang.Math.sqrt; ✓

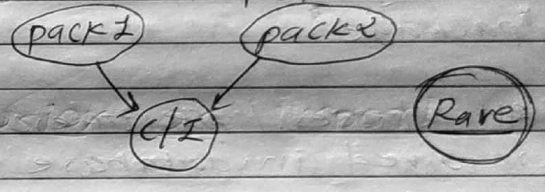
import java.lang; ✗
import java.lang.*; ✓

Date / /
Page

import static java.lang; ✗
import static java.lang.*; ✗

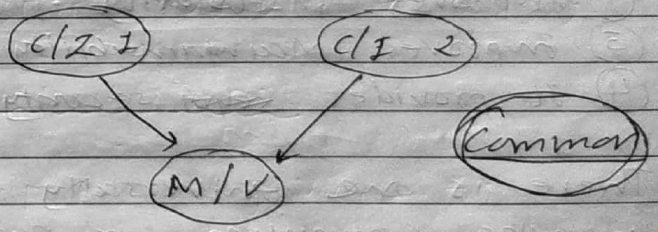
→ Why static import is Not Recommanded?

① Two packages contains a class / Interface with same name is very rare
∴ Ambiguity Problem is very rare in normal import.



BUT

Two classes / Interfaces a variable / Method with same name is very common
∴ Ambiguity problem is very common in static import



(like MAX_VALUE is in Integer, Byte, short ...)

② Usage of static import reduces Readability & creates confusion. Hence if there is no specific requirement, we should not use it.

→ Packages Statements (To pack .class files)

↳ It is an encapsulation mechanism to group related classes / interfaces into a single unit, which is a nothing but "package".

e.g. All classes / interfaces which are required in database operations are grouped into a single package which is [java.sql] package.

↳ The main advantages of package are:
1. To Resolve naming conflict (Unique identification of components)
2. Improves Modularity
3. Improves Maintainability
4. It provides ~~secrit~~ security (As default / package scope is there)

↳ There is one universally accepted naming convention for packages i.e to use Internet Domain Name in reverse.

e.g.

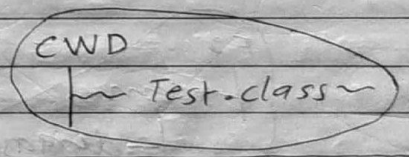
com.icicibank.loan.housing.Account

- com. → client's internet domain name in reverse
- icicibank → module name
- loan → Sub-module name
- housing → class name

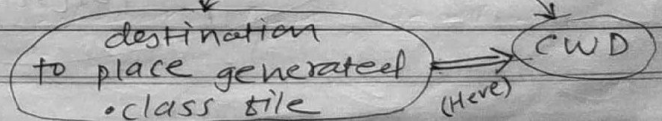
→
```
package com.durgasoft.scjp;

public class Test
{
    p s v main(String[] args)
    {
        s.o.p("pckg demo");
    }
}
```

(i) javac Test.java ↵

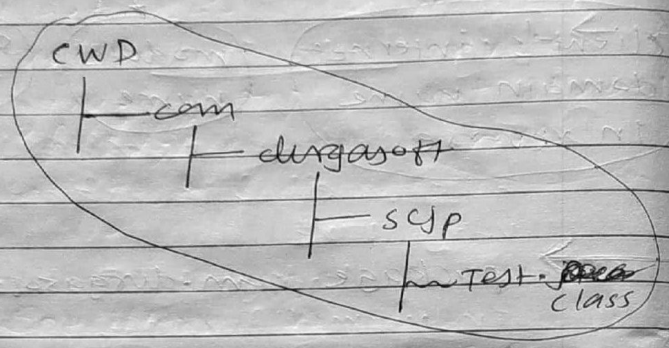
Generated .class file will be placed in ~~current~~ current working directory



(ii) javac -d . Test.java ↵



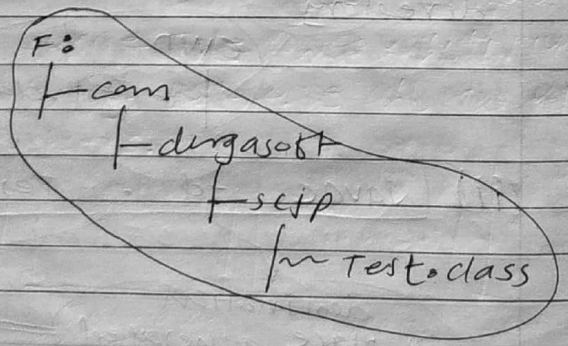
Generated .class file will be placed in corresponding package structure.



⇝ If the corresponding package structure not already available then this command itself create corresponding package structure.

⇝ As destination instead of . (dot), we can take any valid directory name

```
javac -d F: Test.java ↵
```



⇝ If the specified directory not already available, then we will get compile time error.

```
javac -d z: Test.java ↵
```

(Here, z: not avail in my pc)

C.E. directory not found z:

⇝ At the time of execution, we have to use a fully qualified name.

```
java com.durgasoft.sqp.Test ↵
```

O/P ⇒ pckg demo

⇝ In any java source file there can be atmost one package stmt, More than one is not allowed or we will get C.E.

```
package pack1;
package pack2;
public class Test
{
}
```

C.E. class, Interface or enum expected

Date / /
Page

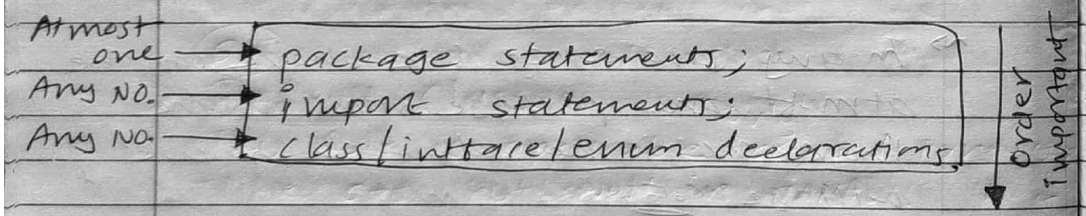
→ In any java program, The first non-comment stmt should be package stmt (if any), otherwise we will get C.E.

```
import java.util.*;
package pack7;
public class Test
{
}
```

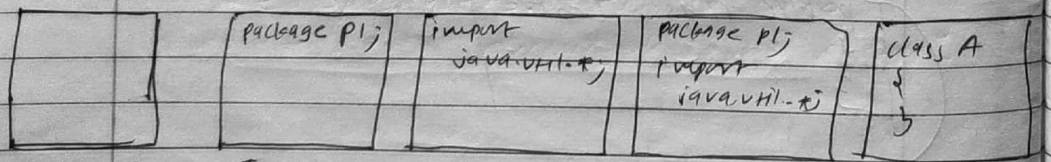
→ C.E. class, interface or enum expected

→ Conclusion on Java Source File

The following is valid java source file structure.



→ An Empty source File is a valid java program. (Hence following all are valid)



(All compiles Fine)

Date / /
Page

(*) Class Level Modifiers

→ Whenever we are writing our own classes we have to provide some information about our class to JVM, like

1. whether this class can be accessable from anywhere or not?
2. whether child class creation is possible or not?
3. whether Object creation is possible or not?

etc.

We can specify this info" by using Appropriate Modifiers.

→ The only Applicable Modifiers for Top Level classes are:

public final strictfp (5)
default abstract

(BUT)

For Inner classes, Applicable Modifiers are:

public final strictfp (8)
default abstract
private static
protected

e.g.

```
private class Test
{
    p s v m(str)
    {
        s.o.p("Hi");
    }
}
```

C.E. modifier private not allowed here

→ Access Specifiers (vs)

Access Modifiers

→ public, private, protected, default are considered as specifiers.
except this
Remainings are considered as Modifiers.

(BUT)

This Rule is applicable only for old languages like C++, but Not in JAVA

In JAVA, All are considered as Modifiers only, there is no word like Specifiers.

→ Public classes

→ If a class declared as a public, then we can access that class from anywhere.

(e.g.)

- ```
  package pack1;
  public class A
  {
      public void m1()
      {
          s.o.p("Hello");
      }
  }
  ```

javac -d . A.java

pack1
A.class

- ```
  package pack2;
  import pack1.A;
  class B
  {
      p s v main(String[] args)
      {
  ```

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

javac -d . B.java
java pack2.B

O/P : Hello

- If class A is Not public then, while compiling B class we will get compile time error,

C.E.
pack1.A is not public in pack1; cannot be accessed from outside package.

→ Default Classes

↳ If a class declared as default, then we can access that class only within current package, i.e from outside package we can't access. Hence, default Access is also known as "package level Access".

→ final Modifier

↳ final is a modifier, applicable for classes, methods & variable

⇒ final Method

Whatever methods parents has, by default available to the child through inheritance,

If the child not satisfied with parent method implementation then child is allowed to redefined the method; based on his requirement

This process is called "overriding".

If the parent class method is declared as final, then we can't override that method in the child class, bcz it's implementation is final.

e.g.

```
class P
{
    public void property()
    {
        S.O.P("Cash + Land + Gold");
    }
    public final void marry()
    {
```

```
        S.o.p("subhalakshmi");
    }
}
class C extends P
{
    public void marry()
    {
        s.o.p("Isha/charvi");
    }
}
```

C.E- marry() in C cannot override marry() in P; overridden method is final

→ final class

↝ If a class declared as final, we can't extend functionality of that class, i.e we can't create child class for that class i.e Inheritance is not possible for final classes.

e.g

```
final class P
{
}
class C extends P
{
}
```

C.E. cannot inherit from final P

↝ Every method present inside final class is final by default, but every variable present inside final class need not be final.

↝ The main Advantage of final keyword is, we can achieve Security & we can provide Unique Implementation.

(BUT)

The Main Dis-advantage is, we are missing key benefits of OOPS: Inheritance [bcz of final classes] & Polymorphism [bcz of final methods]

Hence, if there is no specific requirement then its not recommended to use final keyword.

→ abstract Modifier

↝ abstract is a modifier applicable to classes & Methods but Not for variables.

↝ abstract Method

Even though we don't know about implementation, still we can declare a

Date / /
Page

method with abstract modifier, i.e for abstract methods only declaration is available but (Not) implementation. Hence, abstract method declaration should ends with semicolon(;).

eg. public abstract void m1(); ✓
public abstract void m1(){ } ✗ (C.E.)

Child class is responsible to provide implementation for parent class abstract method.

e.g.

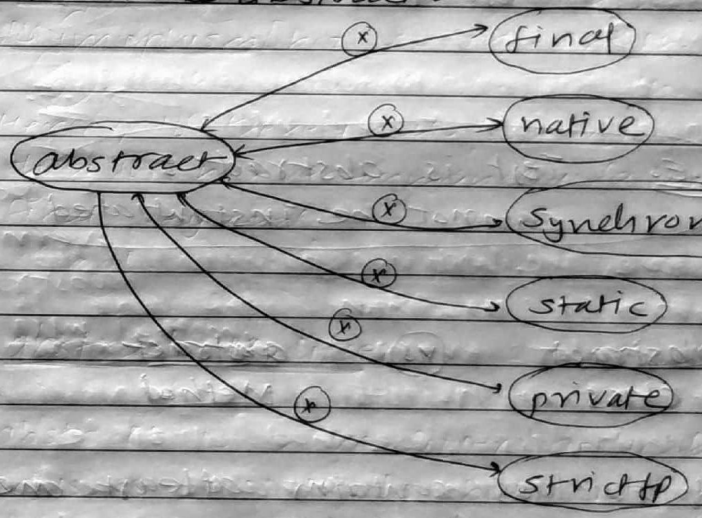
```
abstract class Vehicle
{
    abstract public int getNoOfWheels();
}

class Bus extends Vehicle
{
    public int getNoOfWheels()
    {
        return 6;
    }
}
Class Auto extends Vehicle
{
    public int getNoOfWheels()
    {
        return 3;
    }
}
```

Date / /
Page

→ By declaring abstract method in the parent class, we can provide guidelines to the child classes such that which methods compulsory child has to implement.

→ abstract method never talks about implementation, if any modifier talks about implementation then it forms illegal combinations with abstract.

Following are various illegal combination of modifiers for methods w.r.t abstract.



BCZ these all talks about implementation while abstract should not have implementation for method

abstract final void m1();

C.E: illegal combination of modifiers: abstract & final

Date / /
Page

→ abstract class

→ For any java class, if we are not allowed to create an object (Bcz of partial implementation) such type of class we have to declare with abstract modifier i.e for abstract classes instantiation is not possible

```
abstract class Test
{
    public static void main(String[] args)
    {
        Test t = new Test();
    }
}
```

C.E. Test is abstract : cannot be instantiated

Interview ★

→ abstract class (vs) abstract Method

① If a class contains atleast one abstract method, then compulsory we should declare as abstract, or we will get C.E.

[Reason: If a class contains atleast one abstract method, then implementation is not complete (It is partial) & Hence it is not recommanded to create an object

So, to restrict the object instantiation compulsory we should declare class as abstract]

② Even though class doesn't contain any abstract method, still we can declare class as abstract

[Reason: For any reason, u feel that your implementation is not appropriate or fullfilled & u don't want to instantiate an object]

Abstract class can contain zero (0) no. of methods also.

e.g. ① HttpServlet class is abstract BUT, it doesn't contain any abstract method.

② Every Adapter class is recommanded to declare as abstract, but it doesn't contain any abstract method.

Syntactical Loopholes

-
```
class P
{
    public void m1();
}
```
C.E. missing method body, or declare abstract

-
```
class P
{
    public abstract void m1(){}
}
```
C.E. abstract methods cannot have a body

-
```
class P
{
    public abstract void m1();
}
```
C.E. P is not abstract & does not override abstract method m1() in P

- If we are extending abstract class then for each & every abstract method of parent class, we should provide implementation, otherwise we have to declare child class as abstract.
  In this case next level child class is responsible to provide implementation.

```
abstract class P
{
    public abstract void m1();
    public abstract void m2();
}
class C extends P
{
    public void m1(){}
}
```
C.E. C is not abstract & does not override abstract method m2() in P

final (vs) abstract

(1) abstract methods compulsory we should override to provide implementation
whereas
we can't override final method, Hence final abstract combination is illegal for methods.

(2) For final classes, we can't create child class
whereas
For abstract classes, we should create child class to provide implementation, Hence final abstract combination is also illegal for class.

Interview

③ abstract class can contain final method
whereas
final class can't contain abstract method.

```
abstract class Test
{
    public final void m1()
    {
    }
}
```
✓

```
final class Test
{
    public abstract void m1();
}
```
✗

→ It is highly recommanded to use abstract modifier, because it promotes several OOP features like: Inheritance & Polymorphism.

→ strictfp (strict floating point) modifier

↝ Introduced in 1.2V (IEEE-754 standards)

↝ we can declare strictfp for classes & methods but, Not for variables. (same as abstract)

↝ Usually the result of floating-point arithmetic is varied from platform to platform.

If we want platform-independent for floating-point arithmetic then we should go for strictfp modifier.

Ex. 5.0·D (10.0/3);

→ On 16-bit platform (Processor) result is: 3.333 — (5 or 6 decimal places)

→ On 32-bit platform (Processor) result is: 3.3333 — (14 to 15 decimal places)

↑ without strictfp

↝ strictfp Methods

If a method declared as strictfp all floating-point calculation in that

method has to follow IEEE 754 standard.

So that we will get platform independent result.

```
strictfp public void m1()
{
    s.o.p(10.0/3);  ⟶ IEEE-754
}
```

★ ⇝ abstract modifier never talks about implementations, whereas strictfp method always talks about implementations.

Hence,

abstract strictfp combination is illegal for method.

⇝ **strictfp classes**

If a class declared as strictfp then every floating point calculation present in every **concrete method** has to follow IEEE 754 standards so that we will get platform independent result.

we can declare abstract strictfp combination for classes, that is abstract strictfp combination is

Legal for classes (BUT)

Illegal for Methods.

Because strictfp modifier with class talks only about concrete methods.

while

abstract modifier with that class talks about abstract methods.

abstract strictfp class Test

{

m1(){}

m2(){}

⋮

m100(){}

⟶ IEEE 754

m*();

my();

}

Valid ✓

⇝ abstract strictfp void m1(); ✗ Invalid

(Bcz both talks about body)

Member Level Modifiers

(Method / variable level)

public members

If a member declared as public then we can access that member from anywhere.

BUT

corresponding class should be visible

that is

Before checking member visibility we have to check class visibility.

cwd
- p1
  - A.class

```
package p1;
class A
{
    public void A1()
    {
        s.o.p("A class method");
    }
}
```

java -d . A.java

compiler time

```
package p2;
import p1.A;
class B
{
    p s v m(String[] args)
    {
        A a = new A();
        a.A1();
    }
}
```

java -d . B.java

C.E: p1.A is not public in p1, cannot be accessed from outside package

In the above example, even though my method is public, we can't accessed it from outside package bcz corresponding class is not public

that is

If both class & method are public then only we can access the method from outside package.

default members

If a member declared as default then we can access that member only within the current package that is from outside of package, we can't access.

Hence, default access is also known as package level access.

Private members

If a member is private then we can access that member only within the class, that is from outside of class we can't access.

→ abstract methods should be available to the child classes to provide implementations where as private methods are Not available to child class

Hence,
private abstract combination is ILLEGAL for methods

→ Protected Members

(The most misunderstood modifier in Java)

If a member declared as protected, then we can access that member anywhere within current package but only in child classes of outside package

(protected = default + kids)

→ we can access protected member within the package anywhere either by using parent Reference or by using child Reference.

BUT, access we can't protected in outside package only in child class & we should use child Reference only.

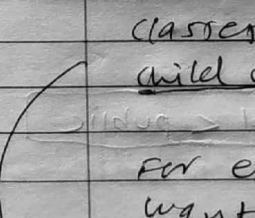
that is
parent reference can't be used to access protected member from outside package

→
```
package pack1;
public class A
{
    protected void m1()
    {
        S.o.p("Hey");
    }
}
class B extends A
{
    p s v m (String[] args)
    {
        ① A a = new A();
          a.m1();

        ② B b = new B();
          b.m1();

        ③ A a = new B();
          a.m1();
    }
}
```

```
package pack2;
import pack1.A;

class C extends A
{
    p s v m (String[] args)
    {
        ① A a = new A();
           a.m1();                    ✗

        ② C c = new C();
           c.m1();                    ✓

        ③ A a1 = new C();
           a1.m1();                   ✗
    }
}
```

C.E.
m1() has protected access in pack1.A

```
package pack2;
import pack1.A;
class C extends A
{
}

class D extends C
{
    p s v m (String[] args)
    {
```



① `A a = new A(); a.m1();`

② `C c = new C(); c.m1();`

③ `D d = new D(); d.m1();` ✓

④ `A a1 = new C(); a1.m1();`

⑤ `A a2 = new D(); a2.m1();`

⑥ `C c1 = new D(); c1.m1();`

C.E.
m1() has protected access in pack1.A

→ We can access protected members from outside package only in child classes & we should use that child class reference only.

For example, from D class if we want to access, we should use D class reference only.

Bcz if That class is default, then we can't use reference of that class like (A a;) etc.

visibility	private	default	protected	public
within same class	✓	✓	✓	✓
From child class of same pack	X	✓	✓	✓
From Non-child of same pack	X	✓	✓	✓
From child class of outside pack	X	X	✓ we have to use that child ref only	✓
From non-child of outside pack	X	X	X	✓

private < default < protected < public

Highly Recommanded ⇒ private For variable (Data Member)

For Member Fun^n ⇒ public

⇝ final variables

① final instance variable

⇒ If the instance variable declared as final, then compulsory we have to perform initialization explicitly, whether we use it or not.

JVM don't provide default values for final instance variable.

Ex:
```
class Test
{
    final int x;
}
```
(X)

(C.E. variable x might not have been initialized)

⇝ **Rule** : For final instance variable, compulsory we should perform initialization before constructor completion.

Date / / Page

↝ Following are various places for initialization:

① At the time of declaration :

```
class Test
{
    final int x=10;
}
```

② Inside instance block :

```
class Test
{
    final int x;
    {
        x=10;
    }
}
```

③ Inside constructor :

```
class Test
{
    final int x;
    Test()
    {
        x=10;
    }
}
```

These are the only possible places to perform initialization of final instance variable.

If we try to do that anywhere else we will get C.E.

Ex.

```
class Test
{
    final int x;
    public void m()
    {
        x=10;
    }
}
```

(X) → C.E. cannot assign value to final variable x

② final static variable

↝ If the static variable declared as final compulsory we should perform initialization explicitly otherwise we will get C.E. whether we use it or Not.

JVM will NOT provide default values to final static variable

```
class Test
{
    final static int x;   (X)
}
```

C.E. variable x might not have been initialized

Rule :- For final static variable, compulsory we should perform initialization befor class-loading completion.

⇝ Following are various places initialization:

① At the time of declaration :-

```
class Test
{
    final static int x=10;
}
```

② Inside static block :-

```
class Test
{
    final static int x;
    static
    {
        x=10;
    }
}
```

⇝ These are the only possible places, if we do it anywhere else we will get CE.

Ex.
```
class Test
{
    final static int x;
    public void m1()
    {
        x=10;   (X)
    }
}
```

C.E. Can-not assign a value to final variable

③ **final local variable**

Non-final local variable

⇝ For simple local variable also JVM won't provide default values.

And,

We will get compile time Error if we use them before initialization.

If we don't use them, we don't need to initialize them.

```
class Test
{
    pub s void main(Str )
    {
        int x;
        S.O.P("Hi");
    }
}
```

o/p = Hii

✓ Valid

Bcz, we don't use x

```
class Test
{
    p s v main(String[] args)
    {
        int x;
        S.o.println(x);   (X) → C.E: variable x might not have been initialized
    }
}
```

~ Even though local variable is final, before using only we have to perform initialization.

And

If we are not using local variable we don't need to initialize it Even though it is final.

∴ For local variable rules are same for final & Non-final

~ The only applicable modifier for local variable is final.

So If we apply any other modifier then we will get C.E.

```
class Test
{
    p s v main(String[] args)
    {
        public int x=10;
        private
        protected
        static
        transient
        volatile
        (✓) final int x;
    }
}
```

C.E: Illegal start of Expression

~ formal parameters of a method can be declared as final,

(they are also local variables to method)

~> static modifier

~ static is a modifier applicable for methods & variables but Not for classes.

we can't declare top-level class with static BUT

we can declare inner class as static (such Inner class ⇒ static Nested class

~ we can't access instance members directly from static area,

BUT

we can access it from instance area directly.

- we can access static members from both instance/static area directly.

~ consider following declarations:

I. int x = 10;

II. static int x = 10;

III. public void m1()
{
S.O.P(x);
}

IV. public static void m1()
{
S.O.P(x);
}

In the same class which combinations can be take simultaneously.

(A) I & III

(B) I & IV ~ (C.E: Non-static variable can't be referenced from static context)

(C) II & III

(D) II & IV

(E) I & II ~ (C.E: var x is already defined in Test)

(F) III & IV ~ (C.E: m1() is already defined in Test)

Both have same signature

[signature = method_name(para_list)]

Case-1

~ Overloading concept applicable for static methods including main(-) method

BUT JVM will call String[] argument method only

```
class Test
{
    public static void main(String[] args)
    {
        S.O.P("String[]");
    }
    public static void main(int[] args)
    {
        S.O.P("int[]");
    }
}
```

(O/P :- String[])

(All other methods we have to call explicitly)

Case-2

~ Inheritance concept applicable for static method including main(-) method.

Hence.

While executing child class if child doesn't contain main(-) then parent class main(-) will executed.

```
class P
{
    p s v main(String[] args)
    {
        s.o.p("parent main");
    }
}
class C extends P
{

}
```

P.java.

Javac P.java ↵

P.class C.class

Java P ↵ parent main

Java C parent main

Case-3

~ class P

```
{
    p s v main(String[] args)
    {
        s.o.p("parent main");
    }
}
```

P.java

It is method-hiding But not over-riding

```
class C
{
    p s v main(String[] args)
    {
        s.o.p("child main");
    }
}
```

Javac P.java ↵

P.class C.class

Java P ↵ parent main

Java C ↵ child main

- It seems overriding concept applicable for static method but its not overriding & it is method hiding.

Note:- For static method overloading & Inheritance is applicable, BUT, overriding is not applicable, Instead of that method hiding is applicable

Method hiding will be in oops concept

Date / /
Page

⇝ Inside method implementation if we are using atleast one instance variable then that method talks about a particular object, Hence we should declare method as instance method

Inside method implementation if we are not using any instance variable, then that method No-where related to particular object, Hence we should declare such method as static method ir-respective of whether we are using static variable or Not.

Ex. class student
{
    string name;
    int rollno;
    int marks;
    static string cName;
};

```
getStudentInfo()
{
    return name+"-"+marks;
}
```
→ Instance method

```
getCollegeInfo()
{
    return cName;
}
```
→ Static method

```
getAverage(int x, int y)
{
    return x+y/2;
}
```
→ Static method

Instance method →
```
getCompleteInfo()
{
    return name+"-"+rollNo+"-"
           +marks+"-"+cName;
}
```

★ ⇝ For static methods, implementation should be available
whereas.
For abstract method, implementation should not be there.

∴ static abstract is illegal combination

⇝ synchronized modifier

⇝ Synchronized is a modifier applicable for methods & blocks but Not for classes & variables.

Date / /
Page

→ If multiple threads trying to operate same java object simultaneously, then there may be a chance of data Inconsistency problem → This is called Race Condition.

→ We can overcome this problem by using synchronized keyword.

→ If a method (or) block delared as synchronized then at a time only one thread allowed to execute that method (or) block on the given object, so that data Inconsistency problem will be resolved.

→ But the main disadvantage of synchronized keyword is it increases the waiting time of threads & it creates performance problems, Hence

If there is no specific requirement then, its not recommended to use synchronized problem.

★ → synchronized talks about implementation method whereas abstract method doesn't contain any implementation.

Hence abstract synchronized is illegal combinations for methods.

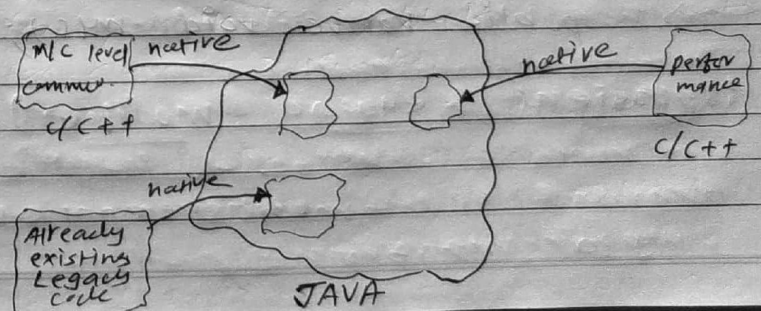
→ native modifier

→ native is a modifier applicable only for methods, & we can't apply any where else.

→ The methods which are implemented in non-java (mostly C or C++) are called native method / foreign methods.

→ The main objectives of native keyword are:

1. To improve performance of system.
2. To achieve machine level (or) memory level communication
3. To use already existing legacy Non-java code.



native abstract ✗
native strictfp ✗

Date / /
Page

~ (Java is not up to the mark in performance wise (comparing with c))

We can fill that gap using native keyword.

~ psvedo code to use native keyword in java.

1. Load native libraries
2. Declare a native method (Not have implementation)
3. Invoke a native method.

```
class NativeDemo
{
    static
    {
①       System.loadLibrary("native library path");
    }
②   public native void m();
}
```

```
class Client
{
    p s v m(String[] args)
    {
        NativeDemo nd = new NativeDemo();
        nd.m();
    }
}
```

~ public native void m(); ✓

~ public native void m(){} ✗

(C.E: native method cannot have a body)

~ For native methods, implementations is already available in other languages.

BUT

For Abstract method, implementation should not be there

Hence

native abstract is Illegal combination of modifier for methods

~ we can't declare native method as strictfp, BCZ there is no guarantee that old languages follow IEEE 754 standard.

Hence

native strictfp is Illegal combination for methods.

~ Native method example :-

```
public native int hashCode();
```

(Bcz, memory operations are there so Java uses this method of other languages)

~ The main Advantage of native keyword is that the performance will be improved BUT

~ The main Dis-Advantage of native keyword is, it breaks platform independent nature of java.
(Bcz C/C++ are platform dependent methods)

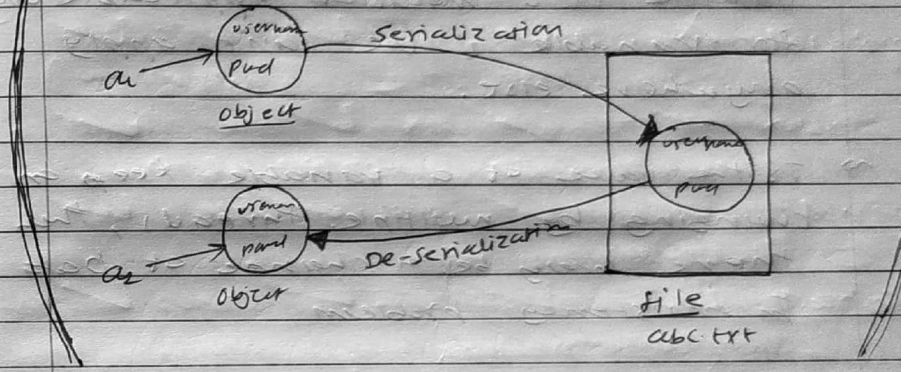
transient keyword (modifier)

(transient = not permanent)

~ transient is a modifier applicable only for variables.

~ We can use transient keyword in serialization context.

means saving the state of an object to file & fetching that object from file is called De-serialization.



★ ~ transient means No to Serialization

~ At the time of serialization if we don't want to save the value of particular variable to meet security constraint, then we should declare ★ that variable as transient.

At the time of serialization, JVM ignores original value of transient variable & save default value to the file. hence

(transient means not to serialize)

→ Here, if we declare pwd as transient, its value in file becomes null.

⇝ Volatile modifier (Volatile = unstable)

⇝ Volatile is a modifier applicable only for variable & we can't apply anywhere else.

⇝ If the value of a variable keep on changing by multiple threads, then there may be a chance of Data Inconsistency problem.

We can solve this problem, By using volatile modifier

⇝ If a variable declared as volatile then for every thread JVM will create a separate local copy.

Every modification performed by a thread will take place in local copy, so that there is no effect on remaining threads.

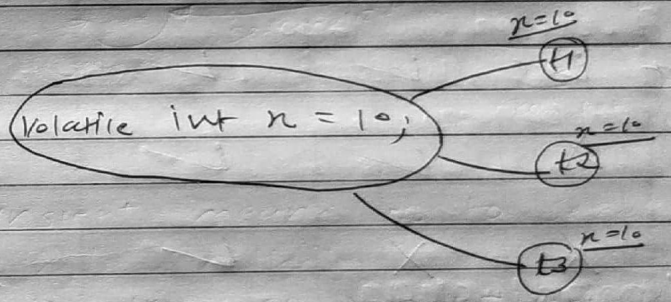
⇝ The main advantage of volatile keyword is we can overcome Data Inconsistency problem,

⇝ BUT, the main disadvantage of volatile keyword is creating & maintaining a separate copy for every thread increases complexity of programming & creates performance problem

Hence if there is no specific requirement, it is never recommended to use volatile keyword

And it is almost deprecated keyword

⇝



⇝ final variable means, the value never changes,

Whereas

volatile variable means, the value keep on changing.

Hence

volatile final is illegal combination for variable.

Modifier	classes		Methods	variables
	outer	Inner		
① public	✓	✓	✓	✓
② private	✗	✓	✓	✓
③ protected	✗	✓	✓	✓
④ <default>	✓	✓	✓	✓
⑤ final	✓	✓	✓	✓
⑥ abstract	✓	✓	✓	✓
⑦ static	✗	✓	✓	✓
⑧ Synchronized	✗	✗	✓	✗
⑨ native	✗	✗	✓	✗
⑩ strictfp	✓	✓	✓	✗
⑪ transient	✗	✗	✗	✓
⑫ volatile	✗	✗	✗	✓

Blocks	Interfaces		Enum		Constructor
	Outer	Inner	Outer	Inner	
✗	✓	✓	✓	✓	✓
✗	✗	✓	✗	✓	✓
✗	✗	✓	✗	✓	✓
✗	✓	✓	✓	✓	✓
✗	✗	✗	✗	✗	✗
✗	✓	✓	✗	✗	✗
✓	✗	✓	✗	✓	✗
✓	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗
✗	✓	✓	✓	✓	✗
✗	✗	✗	✗	✗	✗
✗	✗	✗	✗	✗	✗

Interface

Date / /
Page

1. Introduction
2. Interface Declaration & Implementation
3. extends & implements
4. Interface Methods
5. Interface variables
6. Interface Naming Conflicts
   i) method naming conflicts &
   ii) variable naming conflicts
7. ★ Marker interface (Interview)
8. Adapter classes
9. interface (vs) abstract class
   (vs) concrete class
10. ★ Differences b/w
    interface & abstract class
11. conclusions

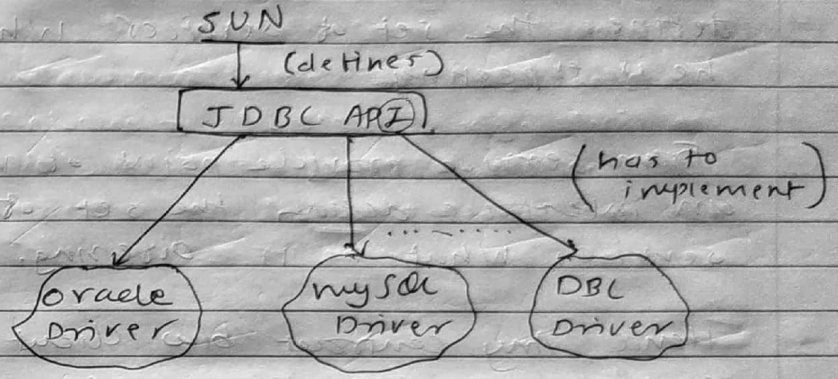
(*) Introduction

→ Definition 1

Any service requirement specification (SRS) is considered as an interface.

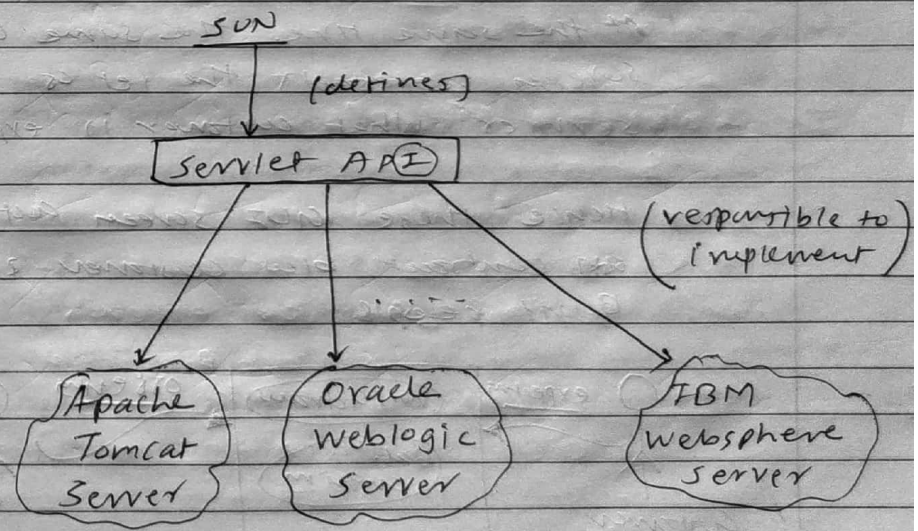
Ex1. JDBC API ~~act as~~ requirement specification to develop database driver.

database vendor is responsible to implement these JDBC API.



Ex2. Servlet API ~~act as~~ requirement specification to develop web-server.

webserver vendor is responsible to implement servlet API



⇝ Definition 2

From client point of view interface defines the set of services what he is expecting.

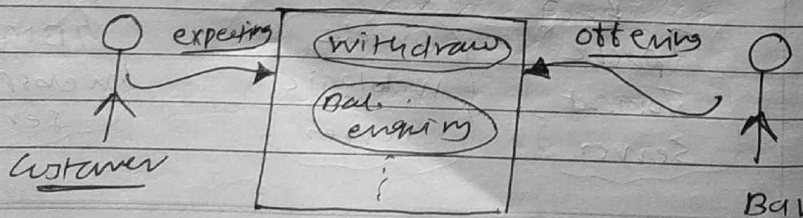
From service provide point of view an interface defines the set of services what he is offering.

Hence, Any contract blw client & service provider is considered as an interface.

Ex. Through Bank ATM GUI screen Bank people are willing the set of services what they are offering

At the same time the same GUI screen represents the set of services what customer is expecting

Hence, these GUI screen act as a contract blw customer & Bank people



Bank ATM GUI Screen

⇝ Definition 3

Inside interface every method is always abstract whether we are declaring or not

Hence,

interface is considered as 100 % pure abstract class.

⇝ Summary Definition

★ Any service requirement specification (or) any contract blw client & service provider (or) 100 % pure abstract class is nothing but interface.

⊛ Interface Declaration & Implementation

Rule ⇝ whenever we are implementing an interface, for each and every method of that interface, we have to provide implementation.

Otherwise we have to declare class as abstract, then next level child class is responsible to provide implementation.

Rule 2

↝ Every interface methods is always public & abstract, wheather we are declaring or not, hence

Whenever we are implementing an interface method, compulsory we should declare as public otherwise we will get C.E.

Ex.

```
interface Intert
{
    void m1();
    void m2();
}

abstract class ServiceProvider
        implements Intert
{
    public void m1()
    {
    }
}

class SubServiceProvider extends
                    ServiceProvider
{
    public void m2()
    {
    }
}
```

✱ extends VS implements

↝ A class can extend only one one class at a time

↝ An interface can extend any no. of interfaces simultaneously

interface A	interface B
{	{
}	}

```
interface C extends A,B
{
}
```

✓

↝ A class can implement any no. of interfaces simultaneously.

↝ A class can extend another class & can implement any no. of interfaces simultaneously

class A extends B implements C,D,E

↝ Which of the following is valid?

(1) A class can extend any no. of classes at a time ✗

(2) A class can implement only one interface at a time ✗

③ An interface can extend only one interface at a time ✗

④ An interface can implement any no. of interface simultaneously ✓

⑤ A class can extend another class or can implement an interface but not both simultaneously ✓

↳ If we want to take both extends & implements together extends should be first.

```
X implements Y extends Z
```
✗

↳ C.E.

Interface Methods

↳ Every method present inside interface is always public & abstract whether we declare it or not.

Ex.
```
interface Intf
{
    void m1();
}
```

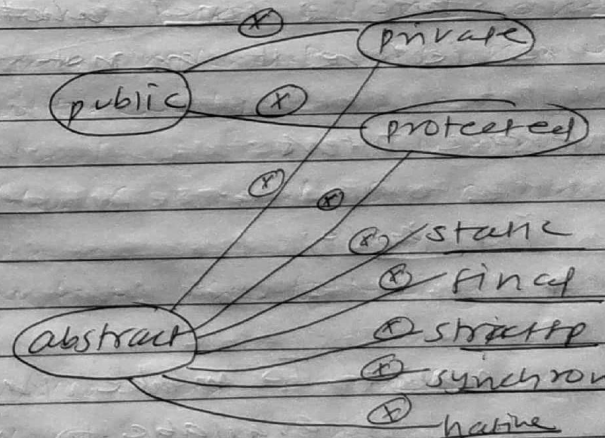
m1() → public :- To make method available to every implementation class.

→ abstract :- Implementation class is responsible to provide implementation.

↳ Hence, inside interface the following declarations are equal.

```
void m1();
public void m1();
abstract void m1();
public abstract void m1();
```

↳ As, Every interface method is public & abstract, we can't declare interface with following modifiers.



Otherwise we will get C.E.

Date / /
Page

↳ which declarations of methods are valid inside interface

public void m1(); ✗
private void m1(); ✗
protected void m1(); ✗
static void m1(); ✗
public abstract native void m1(); ✗
abstract public void m1(); ✓

(*) Interface variables

↳ An interface can contain variables. The main purpose of interface variable is to define requirement level constants.

↳ Every interface variable is always public, static & final whether we are declare it or not

↳ interface Intent
{
    int x=10;
}

→ public :- To make variable available to every implementation class.

static :- Without creating object also, implementation class has to access the variable. (∵ we can't create an object of an interface)

final :- If one implementation class changes value then remaining implementation classes will be affected.
- To restrict this problem.

↳ Hence, within an interface the following variable declarations are equal.

int x=10;
public int x=10;
static int x=10;
final int x=10;
public static int x=10;
public final int x=10;
static final int x=10;
public static final int x=10;

↳ As, every interface variable is always public, static, final we can't declare them with following modifiers.

public
static
final

private
protected
transient
volatile

So, not need for transient ← So, no serialization will happen ← No object creation of interface

~> For interface variables, compulsory we should perform initialization at the time of declaration only, otherwise we will get C.E.

```
interface Intf
{
    int x;
}
```

→ C.E. = expected

Bcz, for final static variable we have to initialize it before class loading, means
↳ at a time of declaration /
↳ In static block
But, In interface there is no static block;

~> Inside implementation class, we can access interface variables, but we can't modify value.

```
interface Intf
{
    int x=10;
}
```

```
class Test
implements Intf
{
    p s v m(Str m)
    {
        x=777;
        s.o.p(x);
    }
}
```

→ C.E. cannot assign value to final variable

```
class Test
implement Intf
{
    p s v m(Str
    {
        int x=7
        s.o.p(x)
    }
}
```

O/P : 777

Interface Naming Conflict

~> Method Naming Conflicts

Case 1 : If two interfaces contains a method with same signature & same return type, then in the

implementation class, we have to provide implementation for only one method.

Ex.

```
interface Intent1                 | interface Intent2
{                                 | {
    public void m1();             |     public void m1();
}                                 | }
```

```
class Test implements Intent1, Intent2
{
    public void m1()
    {
    }
}
```

case 2 : If two interfaces contains a method with same name, but different argument types then in the implementation class we have to provide implementation for both methods & that methods act as overloaded methods.

```
interface L                       | interface R
{                                 | {
    public void m1();             |     public void m1(int i);
}                                 | }
```

```
class Test implements L,R
{
    public void m1()
    {
    }
    public void m1(int i)
    {
    }
}
```

overloaded methods ←

case 3 : If two interfaces contains a method with same signature, but different return types then it is impossible to implement both interfaces simultaneously (If return types are not co-variant)

Ex.

```
interface L                       | interface R
{                                 | {
    public void m1();             |     public int m1
}                                 | }
```

we can't write any java class which implements both interfaces simultaneously.

★ Q. Is a Java-class can implement any number of interface simultaneously?

Ans: Yes, except a particular case which is case-3 in previous page.

Variable Naming Conflicts

↝ Two interfaces can contain a variable with a same name & there may be a chance of variable naming conflicts, But we can solve this problem by using interfaces names.

```
interface L                 | interface R
{                           | {
    int x=777;              |     int x=888;
}                           | }
```

```
Class Test implements L,R
{
    p s v m(String[] args)
    {
        //S.O.P(x);   → C.E. reference to x is ambiguous
        S.O.P(L.x);   777
        S.O.P(R.x);   888
    }
}
```

Interview

★ ✱ Marker Interface

↝ If an interface doesn't contain any methods & by implementing that interface if our object will get some ability.
such type of interfaces are called marker interfaces.

Ex: Serializable (I)
Cloneable (I)
RandomAccess (I)
SingleThreadModel (I)
:
etc

} These are marked for some ability.

These are also known as
"Ability Interface"
(or)
"Tag Interface"

↝ Ex1: By implementing Serializable interface, our objects can be saved to the file & can travel across a network.

↝ Ex2: By implementing Cloneable interface, our objects are in a position to produce exactly duplicate clone objects.

Q. Without having any methods, how the object will get some ability in marker interface?

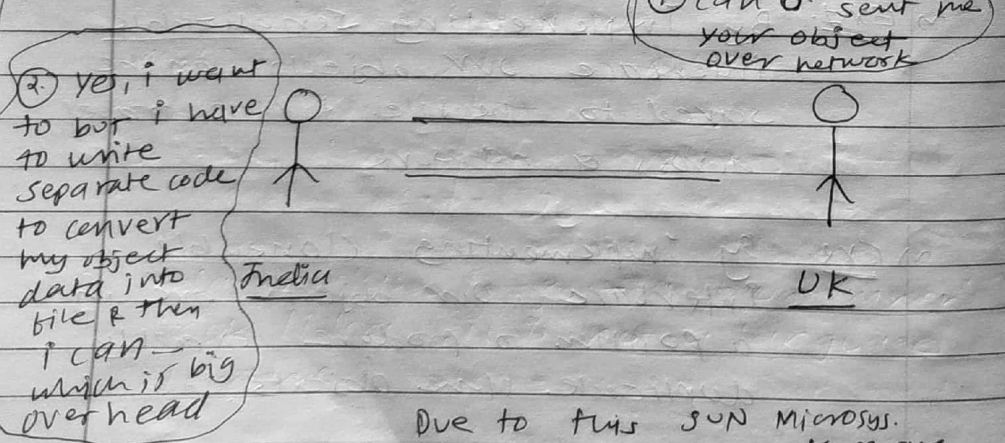
Ans: Internally JVM is responsible to provide required ability.

Q. Why JVM is providing required ability in Marker interface?

Ans: To reduce complexity of programming & to make Java language as simple

Q. Is it possible to create our own Marker interface?

Ans: Yes, but customization of JVM is required.



Due to this SUN Microsys. define Marker interfaces & those are implemented by JVM

Adapter classes

→ Adapter class is simple java class that implements an interface with only empty implementation.

→ problem:

```
interface X
{
    m1();
    m2();
    m3();
    :
    m1000();
}
```

We only need to implement m3() method of interface X but we have to provide implementation for other methods also.

So,

```
class Test implements X
{
    m3()
    {
    }
    m1()
    {
    }
    m2()
    {
    }
    :
    m1000()
    {
    }
}
```

→ 10 lines (required)

→ Extra 1000 lines (Not required, though we have to provide)

This approach increases Length of code & Reduces the Readability

Here, we can use Adapter class, which provides empty implementations of all methods of interface.

We can extend that adapter class for many times & can re-implement only those methods which we need.

```
abstract class AdapterX implements X
{
    m1(){}
    m2(){}
    m3(){}
    :
    m1000(){}
}
```

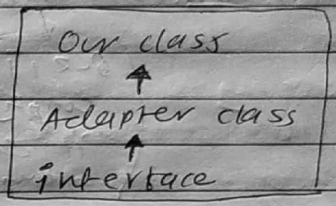
```
class A implements AdapterX
{
    m3()
    {
    ≡
    }
}
```

```
class B implements AdapterX
{
    m1000()
    {
    ≡
    }
}
```

Adapter class is just a programmer's trick/idea, it is not a language feature.

As we don't need to create an object of Adapter class, we should declare that as abstract.

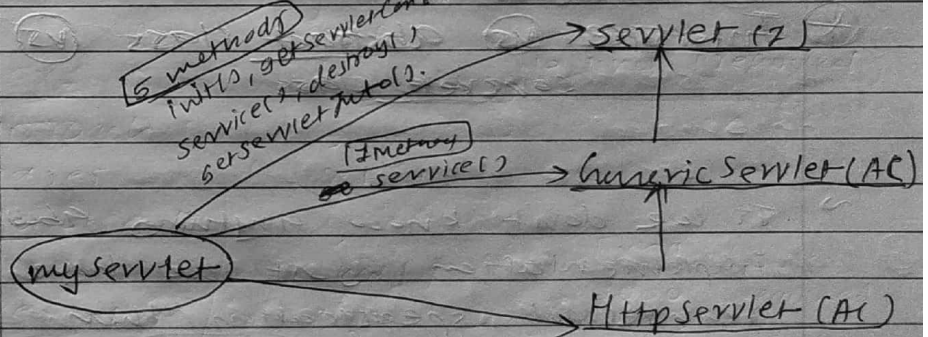
(Bcz it provides only dummy implementation of methods)



Ex.

We can develop a servlet in a following three ways

1) By implementing Servlet (I)
2) By extending GenericServlet (AC)
3) By extending HttpServlet (AC)



If we implement servlet interface, for each & every method of that interface we should provide implementation

LOC (↑), Reusability (↓)

So, we can extend GenericServlet (AC) & now we have to implement only service() method.

∴ So, here GenericServlet (AC) act as Adapter class. (More or Less)

↓

(Bcz, it doesn't contain empty implementation, it contain useful implementation)

Marker Interface & Adapter classes simplifies complexity of programming & these are best utilities to the programmer. & programmers life will become simple.

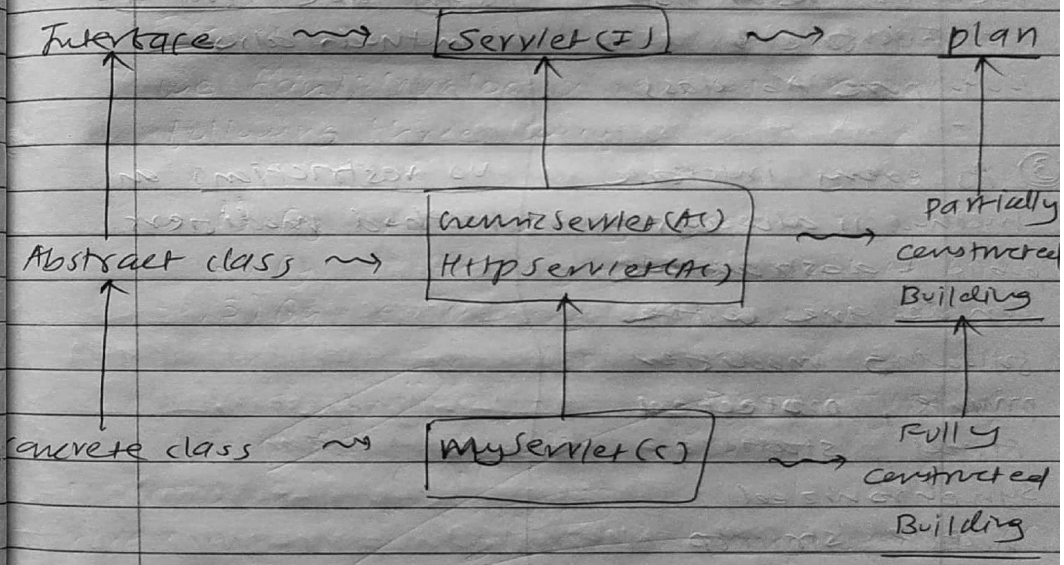
⊛ Interface (vs) abstract class (vs) concrete class

⇝ If we don't know anything about implementation, just we have requirement specification then we should go for (Interface).

⇝ If we are talking about Implementation but not completely (Partial implementation) then we should go for (abstract class).

⇝ If we talking about implementation completely & ready to provide services then we should go for concrete class.

⇝ In other context, we can say that these three are the various phases of any process.



⊛ Differences b/w Interface & Ab class

⇝ Difference table is given

★ Interview

Date ___/___/___
Page ___

Interface	Abstract class
① If we don't know anything about implementation & just we have req. spec., then we should go for interface	If we are talking about partial implementation then we should go for abstract class
② Inside interface every method is public & abstract, whether we declare or not ∴ Interface ∀ 100% pure Ab. class	Every method present inside abstract need not be public & abstract, we can take Concrete methods also
③ As every interface method is always public & abstract we can declare with following modifier private, protected final, static, synchronized, native, strictfp	NO restrictions on method modifiers.
④ Every variable in interface is always public, static, final, whether we are declaring or not.	Every variable in abstract class need not to be public, static final

Interface	Abstract class
⑤ As every interface variable is public, static, final, we can't declare with following modifier: private, protected, volatile, file	No restrictions on abstract class variable modifiers
⑥ For interface variable compulsory we have to perform initialization at the time of declaration only, or we will get C.E. public static final int x=10;	For Abstract class variable we are not required to perform initialization at the time of declaration, we can also do it later on
⑦ Inside interface, we can't declare static & instance blocks ★	we can declare static & instance blocks
⑧ Inside interface, we can't declare constructor ★	we can declare constructor.

Q. Anyway we can't create object of an abstract class, But abstract class can contain constructor, What is the reason?

Ans:- Abstract class constructor will be executed, whenever we creating child class object, to perform initialization of child class object.

Approach 1

⇒ Without having constructor in Abstract class.

```
- abstract class Person
  {
      String name;
      int age;
          ; 100 properties
  }

- class Student extends Person
  {
      int rno;
      Student(String name,
              int age, ... , int rno)
              ← 101 properties →
      {
          this.name = name;
          this.age = age;
              :
          this.rno = rno;
      }
  }
```

100 properties

```
Student s = new Student(101 properties);

- class Teacher extends Person
  {
      String sub;
      Teacher(String name, int age..., String sub)
              ← 101 properties →
      {
          this.name = name;
          this.age = age;
              :
          this.sub = sub;
      }
  }

Teacher t = new Teacher(101 properties);
```

100 properties

- If we have those types of 1000 child classes, we have to initialize all properties of parent in every child class constructor, as parent doesn't contain any constructor to initialize those ones.

So More Code

Code Redundancy will there.

Approach-2

Having Constructor in Abstract class

- abstract class Person
{
    String name;
    int age;
    ; 100 properties

This constructor will work for every child object initialization

    Person(String name, int age, ---)
    |------ 100 properties ------|
    {
        this.age = age;
        this.name = name;
        ;
    }
}

- class Student extends Person
{
    int rno;
    Student(String name, int age --
                int rno)
    |------ 101 properties ------|
    {
        super (100 properties); } 2 LOC
        this.rno = rno;
    }
}
Student s = new Student (101 properties)

- Same for teacher class also.

~> Here, those initialization lines are in Parent class-constructor, so No Redundancy

Q- Anyway we can't create objects for abstract class & interface, But abstract class can contain constructor while interface doesn't contain constr What is the reason?

Ans: The main purpose of constructor is to initialize the instance variable of class.

~> abstract class can contain instance variable, which are required for child object, so to perform initialization of those instance variables, constructor is required.

~> Interface has variable which is always public static final and there is no chance of existing instance variable Hence, constructor is not required.

~> Whenever we are creating child class-object, parent object won't be created, just parent class constructor for the child object purpose only.

Ex.

```
class Parent
{
    Parent()
    {
        s.o.p(this.hashcode());
    }
}
Class Child extends Parent
{
    Child()
    {
        s.o.p(this.hashcode());
    }
}
Class Test
{
    P s v main(String[] args)
    {
        Child c = new Child();
        s.o.p(c.hashcode());
    }
}
```

O/p :

```
11394033
11394033
11394033
```

(So, we proved our statement)

Inside interface every method is always abstract & we can take only abstract methods in abstract class also, then what is the difference blw interface & abstract class? i.e Is is possible to replace Interface with Abstract class.?

Ans : we can replace interface with abstract class, But its not a good programming practice, this something like Recruiting IAS officer for sweeping Activity.

If everything is abstract then it highly recommended to go for interface. but not for abstract class

Approach 1	Approach 2

abstract class X
{
}
 	
class Test extends X
{
}
 	
① While extending abstract class, it is not possible to extends any other class. So, we are missing Inheritance Benefit
	interface X
	{
	}
	
	class Test implements X
	{
	}
	
	While implementing interface, we can extend some other class Hence, we won't miss Inheritance benefit

Date / /
Page

Approach 1	Approach 2
(2) In this case object creation is costly	Here, object creation is not costly
Test t = new Test(); (2 mins)	Test t = new Test(); (2 sec)

End of Declarations & Access Modifiers

Start OOP's Concepts