4. Declarations and Access Modifiers

1. Java Source File Structure
2. class modifiers
3. member modifiers
4. interfaces

1. Java Source File Structure :—

→ A Java program can contain any no. of classes, but atmost one 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 CE.

→ If there is no public class then we can use any name for Java programs and there are no restrictions.

Ex: ①

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

Case(i): If there is no public class then we can use any name for Java program.

Ex: A.java ✓
B.java ✓
C.java ✓
Durga.java ✓

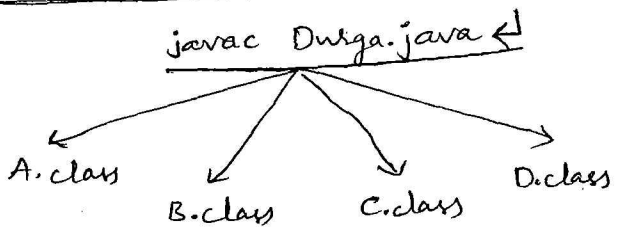
Case(ii): If class B is public then name of the program should be B.java, o.w. we will get CE saying, class B is public, should be declared in a file named B.java.

Case(iii): If both B and C classes declared as public & name of the program is B.java then we will get CE saying, class C is public,

should be declared in a file named C.java.

Ex②:

```
class A
{
  p s v m(-)
  {
    S.o.p("A class main");
  }
}
class B
{
  p s v m(-)
  {
    S.o.p("B class main");
  }
}
class C
{
  p s v m(-)
  {
    S.o.p("C class main");
  }
}
class D
{ }
```



java A
O/P: A class main

java B
O/P: B class main

java C
O/P: C class main

java D
RE: NoSuchMethodError: main

java Durga
RE: NoClassDefFoundError: Durga

Conclusions:—

1. We can compile a Java program, but we can run Java class.
2. Whenever we are compiling a Java program for every class one .class file will be created.
3. Whenever we are executing a Java class that corresponding class main(-) method will be executed.
4. If the class doesn't contain main(-) method then we will get RE saying, NoSuchMethodError: main.
5. Whenever we are executing a Java class if the corresponding .class file is not available then we will get RE saying, NoClassDefFoundError.
6. It is highly recommended to take only one class for source file & name of the program and name of the class must be matched.

The advantage of this approach is readability & maintainability will be improved.

import statement :—

Ex:
```
class Test
{
  P S V m(-)
  {
    ArrayList l=new ArrayList();
  }
}
```
→ CE: cannot find symbol
symbol : class Arraylist
location: class Test

→ We can resolve this problem by using fully qualified name.

```
java.util.AL l=new java.util.AL();
```
→ Fully qualified name.

→ The problem with usage of fully qualified name every time is it increases length of the code & reduces readability.

→ We can solve this problem by using import statement.

Ex:
```
import java.util.AL;
class Test
{
  P S V m(-)
  {
    AL l=new AL();
  }
}
```
→ short name

→ Whenever we are using import statement we are not required to use fully qualified name & we can use short names directly.

→ Hence import statement acts as Best typing shortcut.

Case(i): Types of import statements :—

→ There are 2 types of import statements.

1. Explicit class import
2. Implicit class import.

1. Explicit class import :—

Ex: import java.util.AL;

→ It is highly recommended to use explicit class import becoz it improves readability.

→ Best suitable for Hitech city where readability is important.

2. Implicit class import :—

Ex: import java.util.*; implicit class

→ It is not recommended to use import becoz it reduces readability of the code.

→ Best suitable for Ameerpet where typing is important.

Case(ii): Which of the following import statements are meaningful :—

✓ ① import java.util.*;

✗ ② import java.util.AL.*;

✗ ③ import java.util;

✓ ④ import java.util.AL;

Case(iii): Consider the following class :—

Ex: class MyObject extends java.rmi.UniCastRemoteObject
{ }

→ The code compiles fine eventhough we are not using import statement becoz we used fully qualified name.

Note:— Whenever we are using fully qualified name it is not required to write import statement.

. Ily whenever we are writing import statement then it is not required to use fully qualified name and we can use short names directly.

Case(iv):

Ex: import java.util.*;
import java.sql.*;
class Test
{
p s v m(—)

```
{
  Date d=new Date();  ---> CE: reference to Date is ambiguous
}
```

*** Note:- We will get same ambiguity problem even in case of List also beeoz it is available in both util and awt packages.

*** Case(v):

→ While resolving class names compiler will always give the precedence in the following order.

1. Explicit class import
2. Classes present in Current Working Directory (CWD).
3. Implicit class import.

Ex:
```
import java.util.Date;
import java.sql.*;
class Test
{
  p s v m(-)
  {
    Date d=new Date();
  }
}
```

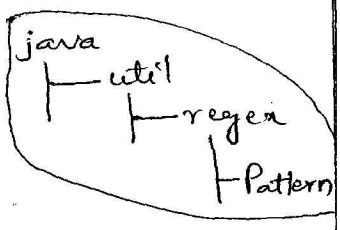
→ The code compiles fine and util package Date will be considered.

Case(vi):

→ Whenever we are importing a package all classes & interfaces present in that package are by default available, but not subpackage classes.

→ Hence to use sub package class compulsory we should write import statement until sub package level.

Ex: We use Pattern class in our program which import statement we have to use:



✗ ① import java.*;
✗ ② import java.util;

✓ ③. import java.util.regex.*;

✗ ④. no import is required

***

Case(vii):

→ The following 2 packages are not required to import becoz all classes & interfaces present in these packages are by default available in every Java program.

① java.lang package

② default package (Current Working Directory).

Case(viii):

→ import statement is totally compile time issue & there is no effect on execution time.

→ Hence if more no. of imports present then more will be the compile time.

***

Case (ix): Difference btw C language #include and Java language import statement:—

#include<stdio.h>	import java.io.*;
1. In this case, all specified header files will be included at the time of compilation only.	1. Specified library classes wont be included at the compile time. and at runtime JVM will go to corresponding library class & execute and place the result in our program.
2. Memory wastage is high.	2. Memory wastage is low.
3. Relatively performance is high.	3. Relatively performance is low.
4. It is static import (translation time import).	4. It is dynamic import (execution/ runtime import).

Note:- 1.5 version new features

1. for-each loop
2. var-arg method
3. co-varient return types
4. Autoboxing & Autounboxing
5. Queue
6. Generics
7. enum
8. Annotations
9. static import

9. Static import:-

→ Introduced in 1.5 version.

→ According to SUN people static import improves readability of the code.

→ According to World wide programming experts (like us) usage of static import reduces readability and creates confusion.

→ Hence if there is no specific requirement then it is not recommended to use static import.

→ Usually we can access static members by using class name, but whenever we are writing static import then we are not required to use class names and we can access static members directly.

Ex: Without static import

```
class Test
{
  p s v m(-)
  {
    S.o.p(Math.sqrt(4));
    S.o.p(Math.random());
  }S.o.p(Math.max(10,20));
}
```

With static import

```
import static java.lang.Math.sqrt;
import static java.lang.Math.*;
class Test
{
  p s v m(-)
  {
    S.o.p(sqrt(4));
    S.o.p(random());
```

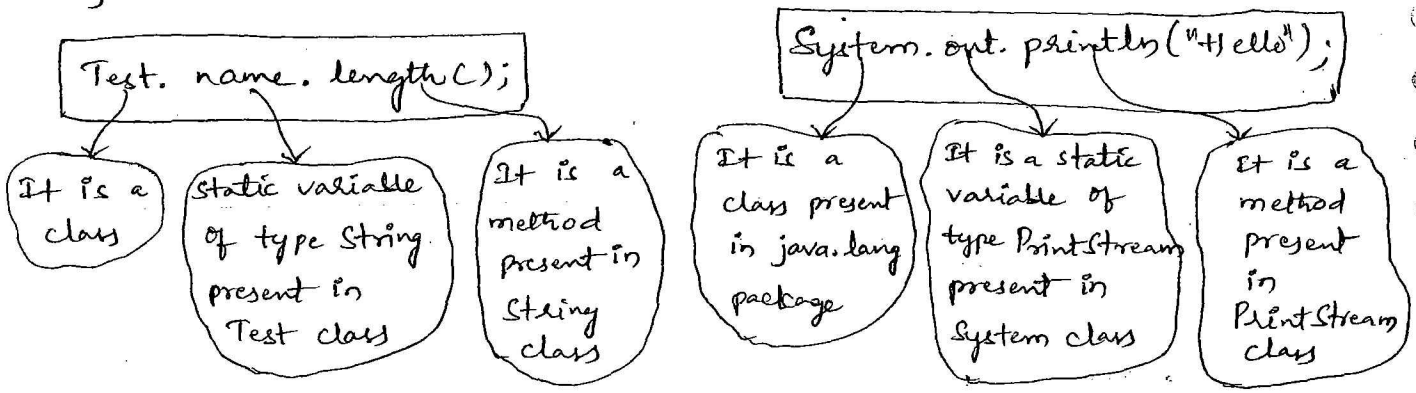
```
        S.o.p(max(10, 20));
    }
}
```

***

Q: Explain about System.out.println:-

Ans:
```
class Test
{
    static String name="java";
}
```

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



→ As out is a static variable present in System class, whenever we are writing static import we can access out directly without class name.

Ex:
```
import static java.lang.System.out;
class Test
{
    p s v m(-)
    {
        out.println("Hello");
        out.println("Hi");
    }
}
```

Ex: ①
```
import static java.lang.Byte.*;
import static java.lang.Integer.*;
class Test
{
    p s v m(-)
    {
        S.o.p(MAX_VALUE);
    }
}
```
→ CE: reference to MAX_VALUE is ambiguous

Ex ②: import static j.l.Byte.MAX_VALUE; ——→ ②
import static j.l.Integer.*;
class Test
{
static int MAX_VALUE = 999; ——→ ①
P S V m(—)
{
S.o.p(MAX_VALUE); ⇒ o/p: 999
}
}

→ While resolving static members compiler will always give the precedence in the following order.

1. Current class static members
2. Explicit static import
3. Implicit static import

→ If we comment line ① then explicit static import will be considered and the o/p is 127, which is Byte class MAX_VALUE.

→ If we comment lines ① & ② then Integer class MAX_VALUE will be considered, which is 2147483647.

Normal import syntax:—

```
import packagename.classname;
import packagename.*;
```

Static import syntax:—

```
import static packagename.classname.static_members;
import static packagename.classname.*;
```

Q: Which of the following import statements are valid?

✗ ① import java.lang;
✓ ② import java.lang.Math;
✗ ③ import java.lang.Math.sqrt;

✓④ import java.lang.*;
✗⑤ import java.lang.Math.*;
✗⑥ import java.lang.Math.sqrt.*;
✗⑦ import static java.lang;
✗⑧ import static java.lang.Math;
✓⑨ import static java.lang.Math.sqrt;
✗⑩ import static java.lang.Math.sqrt();
✓⑪ import static java.lang.Math.*;
✗⑫ import static java.lang.Math.sqrt.*;
✗⑬ import static java.lang.Math;

Note:- Two packages contain a class or interface with the same name is very rare and hence ambiguity problem is also very rare in normal import.

But 2 classes contain a variable or method with same name is very common & hence ambiguity problem is also very common in static import.

Moreover usage of static import creates confusion & reduces readability. Hence if there is no specific requirement then it is never recommended to use static import.

***Difference b/w Normal import and static import:-

→ We can use normal import to import classes & interfaces of a package.
→ Whenever we are using normal import it is not required to use fully qualified name and we can access classes directly by using short names.
→ We can use static import to import static members of a class.
→ Whenever we are using static import we can access static members directly without class name.

Package :—

→ It is an encapsulation mechanism to group related classes & interfaces into a single module.

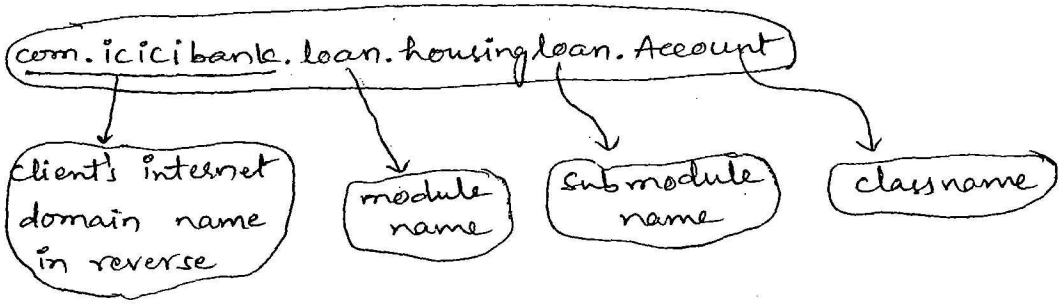
Ex ①: All classes & interfaces which are required for database Operations are grouped into a separate package, which is nothing but sql package.

Ex ②: All classes & interfaces which are required for File I/O Operations are grouped into a separate package, which is nothing but io package.

→ The main advantages of package statement are,

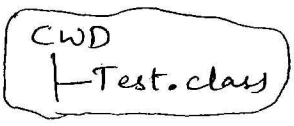
1. We can resolve naming conflicts.
2. It improves maintainability of the application.
3. It improves modularity of the application
4. It provides security.

→ There is one universally accepted naming convention for packages i.e., to use internet domain in reverse.



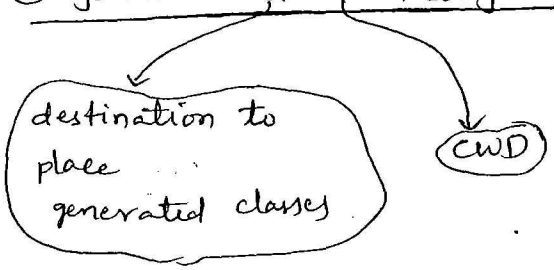
Ex:
```
package com.durgasoft.scjp;
public class Test
{
  p s v m(-)
  {
    S.o.p("package Demo");
  }
}
```

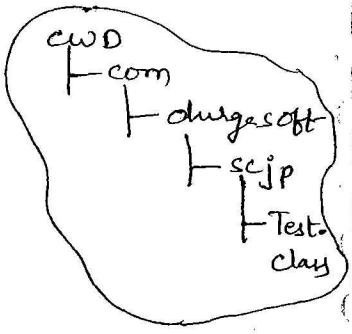
① javac Test.java ↵



→ Generated .class file will be placed in CWD

② javac -d . Test.java ↵



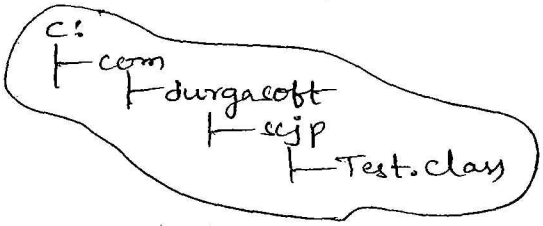


→ Generated classes will be placed in corresponding package structure.

→ If the required package structure is not already available then the above command itself will create that folder structure.

→ Instead of dot(.) we can use any valid directory name.

Ex: javac -d c: Test.java ↵



→ If the specified destination is not already available then we will get CE.

Ex: javac -d z: Test.java ↵

If z: is not available then we will get CE.

→ At the time of execution compulsory we have to specified fully qualified name.

Ex: java com.durgasoft.scjp.Test ↵ O/P: package Demo.

***

Conclusions :-

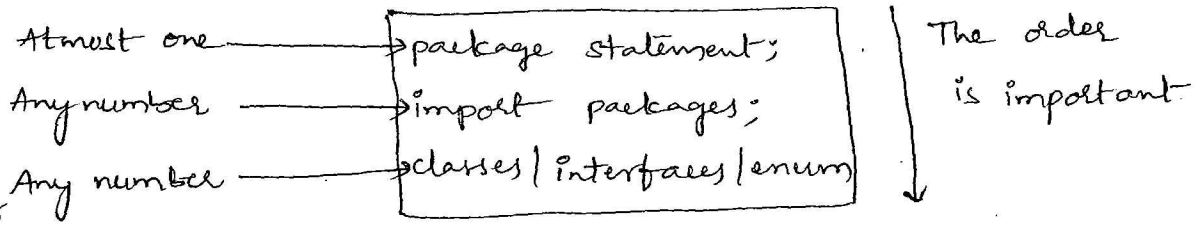
1. In any Java program, there should be atmost one package statement i.e., we can't take more than one package statement.

Ex: package pack1;
package pack2;
class A
{ }
→ CE: class, interface or enum expected

2. In any Java program, first non-comment statement should be package statement (if it is available).

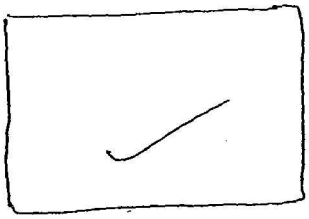
Ex:- import java.util.AL;
package pack1;
public class A
{ }
→ CE: class, interface or enum expected

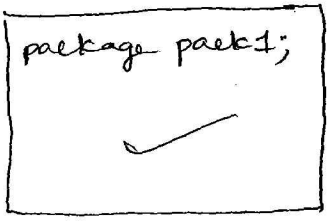
→ The following is the valid Java File Structure.



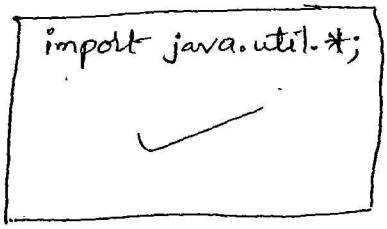
*** Note:- An empty source file is a valid Java programs.

→ The following are valid Java programs.

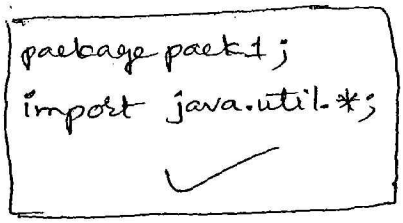

Test.java



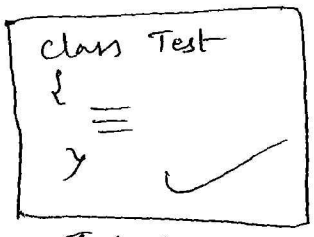
Test.java



Test.java



Test.java



Test.java

2. Class modifiers :—

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

→ This information including

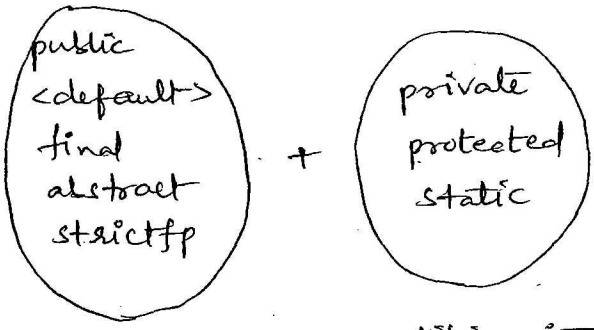
1. Whether this class can be accessible from anywhere or not
2. Whether child class creation is possible or not.
3. Whether instantiation is possible or not

→ We can provide this information by declaring with appropriate modifiers.

→ The only applicable modifiers for top level classes are

public
<default>
final
abstract
strictfp

→ But for inner classes applicable modifiers are



Access specifiers Vs modifiers :—

→ private, public, <default> & protected are considered as access specifiers and except this all remaining are considered as access modifiers.

→ But this terminology is applicable only for old languages like C, C++ and so on.

→ But in Java there is no such type of terminology. All are considered as modifiers only.

Ex: private class A
{ }
CE: modifier private not allowed here.

public classes:—

→ If a class declared as public then we can access that class from anywhere either within or outside of package.

Ex:
```
package pack1;
public class A
{
  public void m1()
  {
    S.o.p("A class method");
  }
}
```
javac -d . A.java ↵ ✓

pack1
└ A.class

```
package pack2;
import pack1.A;
class B
{
  p s v m(-)
  {
    A a=new A();
    a.m1();
  }
}
```
javac -d . B.java ↵ ✓
java pack2.B ↵ ✓
O/P : A class method

→ If class A is not public then while compiling B class we will get CE saying,

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

default classes:—

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

final :-

→ final is the modifier applicable for classes, methods and variables.

final method :-

→ Whatever parent has by default available to the child through inheritance.

→ If the child is not satisfied with Parent implementation then child is allowed to override that method based on its requirement.

→ If the Parent class method declared as final then child is not allowed to override that method i.e., we can't override final methods.

Ex:-
```
class P
{
  public void property()
  {
    S.o.p("cash+land+gold");
  }
  public final void marry()
  {
    S.o.p("Subbalakshmi");
  }
}
```

```
class C extends P
{
  public void marry()
  {
    S.o.p("3sha/9tara/4me");
  }
}
```

CE: marry() in C cannot override marry() in P; overridden method is final

final class :-

→ If a class declared as final then we can't create child class i.e., we can't extend the f'ality.

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

CE: Cannot inherit from final P

Note:- Every method present inside final class is always final whether we are declaring or not.

But every variable present inside final class need not be final.

→ The main advantage of final keyword is we can achieve Security as no one is allowed to change or extend our f°ality

→ But the main disadvantage of final keyword is we are missing Key benefits of OOPS: inheritance (by final classes), polymorphism (by final methods).

→ If there is no specific requirement then it is never recommended to use final keyword.

abstract modifier:-

→ abstract modifier applicable only for methods & classes, but not for variables.

abstract method:-

→ Eventhough we don't know about implementation still we can declare a method with abstract modifier i.e., abstract method has only declaration but not implementation.

→ Hence abstract method declaration should compulsory ends with semicolon.

Ex:- ✓public abstract void m1();
✗public abstract void m1(){}

→ Child class is responsible to provide implementation for Parent class abstract methods.

Ex:
```
abstract class Vechicle
{
  public abstract int getNoOfWheels();
}
```

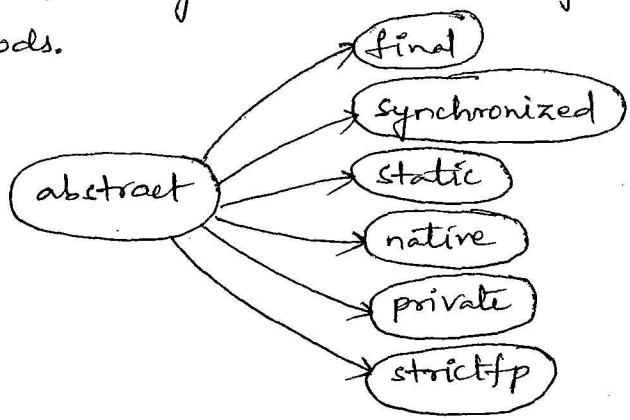
```
class Bus extends Vechicle
{
  public int getNoOfWheels()
  { return 7;
  }
}
```

```
class Auto extends Vechicle
{
  public int getNoOfWheels()
  { return 3;
  }
}
```

→ The main advantage of placing abstract method in the parent class is we can provide guidelines to the child classes such that which method compulsory child has to implement.

*** → abstract method never talks about implementation. If any method talks about implementation then that method forms illegal combination with abstract modifier.

→ Hence the following are various illegal combinations of modifier for methods.



Ex: public abstract final void m1();

→ CE: illegal combination of modifiers: abstract and final

abstract class:—

→ For any Java class, if we don't want object creation such type of class we have to declare with abstract modifier i.e., for abstract classes instantiation is not possible.

Ex:
```
abstract class Test
{
  p s v m(-)
  { Test t=new Test();
  }
}
```

CE: Test is abstract; cannot be instantiated

abstract class Vs abstract method :—

1. If a class contains atleast one abstract method then implementation is not complete. Hence we are not allowed to create object.

   Due to this we should declare class as abstract i.e., if a class contains atleast one abstract method then compulsory we should declare class as abstract, o.w. we will get CE.

2. Eventhough class doesn't contain any abstract methods still we can declare class as abstract if we don't want instantiation for that class. i.e., abstract class can contain 0(zero) no. of abstract methods.

Ex: ① HttpServlet class is abstract class, but it doesn't contain any abstract methods.

Ex: ② Every Adapter class is an abstract, but it doesn't contain any abstract methods.

abstract Vs final :—

→ For abstract methods we should override in child class to provide implementation, but for final methods we can't override. Hence final, abstract combination is illegal for methods.

→ For abstract classes we should create child class to provide proper implementation, but for final classes we can't create child class. Hence abstract final combination is illegal for classes.

→ We can't declare abstract method inside final class, but we can declare final method inside abstract class.

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

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

Ex①:
```
class P
{
  public void m1();
}
```
→ CE: missing method body, or declare abstract

Ex②:
```
class P
{
  public abstract void m1(){ }
}
```
→ CE: abstract methods cannot have a body

Ex③:
```
class P
{
  public abstract void m1();
}
```
CE: P is not abstract and does not override method m1() in P

→ If we are extending any abstract class then for each and every abstract method of parent class we should provide implementation, o.w. we have to declare child class as abstract.

Ex:
```
abstract class P
{
  public abstract void m1();
  public abstract void m2();
}
class C extends P
{
  public void m1(){ }
}
```
CE: C is not abstract and does not override abstract method m2() in P

Note:- It is always a good programming practice to use abstract modifier in our programs becoz it promotes several OOP features.

Strictfp (strict floating point):-

→ Introduced in 1.2 version.

→ strictfp modifier applicable for classes and methods, but not for variables.

strictfp method :—

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

→ If a method declared as strictfp all floating point calculations in that method has to follow IEEE 754 standard so that we will get platform independent results.

strictfp class :—

→ If a class declared as strictfp then every concrete method in that class has to follow IEEE 754 standard so that we will get platform independent results.

*** strictfp Vs abstract :—

→ strictfp method talks about implementation where as abstract method never talks about implementation.

Hence abstract strictfp combination is illegal for methods.

→ But we can declare abstract class as strictfp i.e., abstract strictfp combination is legal for classes, but illegal for methods.

Ex: ✓ `abstract strictfp class Test { }`

✗ `abstract strictfp void m1();`

CE: illegal combination of modifiers: abstract & strictfp

3. member modifiers :—

1. public members :—

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

→ But the corresponding class should be visible i.e., before checking member visibility we have to check class visibility.

→ If both class & members are public then only we can access that members from outside package.

Ex:

```
package pack1;
class A
{
  public void m1()
  {
    S.o.p("A class method");
  }
}
```

pack1 ← A.class

javac -d . A.java ✓

```
package pack2;
import pack1.A
class B
{
  p s v m(-)
  {
    A a=new A();
    a.m1();
  }
}
```

javac -d . B.java ✗

→ In the above example, eventhough m1() method is public we can't access from outside package becoz the corresponding class A is not public i.e., if both method and class are public then only we can access that method from outside package.

2. default members :—

→ If a member declared as default then we can access that member within the current package anywhere i.e., from outside package we can't access.

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

3. private members :—

→ If a member declared as private then we can access that member only within the current class i.e., from outside the class we can't access.

→ abstract method should be visible in the child classes to provide implementation where as private methods are not visible to the child classes.

*** Hence private abstract combination is illegal for methods.

4. protected members :—

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

protected = <default> + kids

*** → We can access protected members within the current package anywhere either by using parent reference or child reference.

*** → But outside package we can access protected members only by using child reference & we can't use parent reference to access protected members from outside package.

Ex:

```
package pack1;
public class A
{
  protected void m1()
  {
    S.o.p("misunderstood method");
  }
}
class B extends A
{
  P S V m(-)
  {
    A a=new A();
    a.m1();
    B b=new B();
    b.m1();
    A a1=new B();
    a1.m1();
  }
}
class D extends C
{
  P S V m(-)
  {
    A a=new A();
    a.m1();
    C c=new C();
    c.m1();
    D d=new D();
    d.m1();
  }
}
```

```
package pack2;
import pack1.A;
class C extends A
{
  P S V m(-)
  {
    A a=new A();
    a.m1();
    C c=new C();
    c.m1();
    A a1=new C();
    a1.m1();
  }
}
A a1=new D();
a1.m1();
C c1=new D();
c1.m1();
A a2=new C();
a2.m1();
}
```

CE: m1() has protected access in pack1.A

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

For Example, To access from C class we should use C class reference and to access from D class we should use D class reference.

*** Note:- Object class contains 2 protected methods clone() and finalize() methods. While accessing these methods we have to take special care.

*** Summary of public, <default>, protected and private modifiers:-

Visibility	private	<default>	protected	public
1. Within the same class	✓	✓	✓	✓
2. From child class of same package	✗	✓	✓	✓
3. From non-child class of same package	✗	✓	✓	✓
4. From child class of outside package	✗	✗	✓ [we should use corresponding child class reference only]	✓
5. From non-child class of outside package	✗	✗	✗	✓

→ The most restricted modifier is private and the most accessible modifier is public.

private < default < protected < public

→ Recommended modifier for variable is private where as for methods is public.

*** final variable :—

1. final instance variable :—

→ If the value of a variable is varied from object to object such type of variables are called instance variables.

→ For every object a separate copy of instance variable will be created.

→ For the instance variables, we are not required to perform initialization explicitly, JVM will provide default value.

Ex:
```
class Test
{
  int x;
  P S V m(-)
  {
    Test t=new Test();
    S.O.p(t.x); ⇒ O/P : 0
  }
}
```

→ If the instance variable declared as final then compulsory we have to perform initialization explicitly whether we are using or not, o.w. we will get CE.

Ex:
```
class Test
{
  final int x;
}
```
CE: variable x might not have been initialized

Rule:-

→ For final instance variables, compulsory we have to perform initialization before constructor completion. i.e., the following are various possible places to perform initialization for final instance variables.

1. At the time of declaration :—

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

2. Inside instance block :—

Ex:
```
class Test
{
  final int x;
  {
    x=10;
  }
}
```
→ instance block

3. Inside constructor :—

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

→ These are the only places to perform initialization for instance variables.

→ If we perform initialization anywhere else we will get CE.

Ex:
```
class Test
{
  final int x;
  public void m1()
  {
    x=10;
  }
}
```
CE: cannot assign a value to final variable x

2. final static variable :—

→ If the value of a variable is not varied from object to object then it is never recommended to declare that variable as instance variable. we have to declare such type of variables at class level by using static modifier.

→ In case of instance variable, for every object a separate copy will be created, but in case of static variables a single copy will be created at class level & shared by every object of that class.

→ For the static variables, we are not required to perform initialization explicitly & JVM will provide default value.

Ex:
```
class Test
{
  static double d;
  p s v m()
  {
    S.o.p(d); ⇒ O/P : 0.0
  }
}
```

→ If we declare static variable as final then compulsory we should perform initialization explicitly, o.w. we will get CE.

Ex:
```
class Test
{
 final static double d;
}
```
→ CE: variable d might not have been initialized

Rule:-

→ For final static variables, compulsory we should perform initialization before class loading completion i.e., the following are various places to perform initialization for final static variables.

1. At the time of declaration:-

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

2. Inside static block:-

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

→ These are the only possible places to perform initialization for final static variables.

→ If we perform initialization anywhere else we will get CE.

Ex:
```
class Test
{
 final static int x;
 P s v m(-)
 {
  x=10;
 }
}
```
→ CE: cannot assign a value to final variable x

3. final local variable:-

→ Sometimes to meet temporary requirements of the programmer we have to delare variables inside a method / block / Constructor such type of variables are called local variables.

→ For the local variables, JVM won't provide any default values. Compulsory we have to perform initialization explicitly, before using that local variable.

Ex:
```
class Test
{
 P s v m(-)
```

```
class Test
{
 P s v m(-)
```

```
{
  int x;
  S.o.p("Hello");
}
```

O/P : Hello

```
{ int x;
  S.o.p(x);
}
```

CE: variable x might not have been initialized

→ Eventhough local variable declared as final before using only we have to perform initialization.

*** → If we are not using a local variable then it is not required to perform initialization eventhough it is final.

Ex:
```
class Test
{
  p s v m(-)
  {
    final int x;
    S.o.p("Hello");
  }
}
```

O/P : Hello

```
class Test
{
  p s v m(-)
  {
    final int x;
    S.o.p(x);
  }
}
```

CE: variable x might not have been initialized

*** Note:— The only applicable modifier for local variables is final. By mistake if we are toying to declare with any other modifier then we will get CE.

Ex:
```
class Test
{
  p s v m(-)
  {
    public int x=10;
    private int x=10;
    protected int x=10;
    static int x=10;
    transient int x=10;
    volatile int x=10;
    final int x=10;
  }
}
```

CE: Illegal start of expression

Note:- If we are not declaring any modifier then it is by default <default> modifier, but this rule is applicable only for instance & static variable but not for local variables.

→ Formal Parameters :-

→ Formal parameters of a method simply acts as local variables of that method.

→ Hence we can declare formal parameter as final.

→ If the formal parameter declared as final then within the method we can't change it's value.

→
```
class Test
{
  p s v m(-)
  {
    m1(10, 20);        → Actual parameters
  }
  p s v m1(final int x, int y)   → Formal parameters
  {
    //x=100;   → CE: cannot assign a value to the final variable
    y=200;
    S.o.p(x+"..."+y);
  }
}
```
O/P : 10 ... 200

*** Conclusions :-

→ For instance & static variables JVM will always provide default values & we are not required to perform initialization explicitly.

→ But if the instance & static variables declared as final then JVM won't provide default values compulsory we have to perform initialization explicitly.

→ But for local variables JVM won't provide default values compulsory we should perform initialization explicitly before using that variable. This rule is same whether local variable is final or not.

Static modifier:—

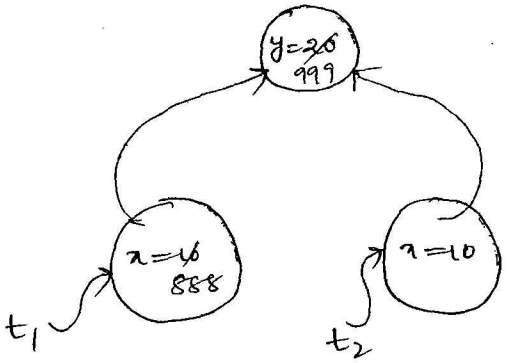
→ static is the modifier applicable only for variables & methods but not for classes.

→ We can't declare top level class as static, but we can declare inner class as static (Static Nested classes).

→ In case of instance variable, for every object a separate copy will be created, but in case of static variable a single copy will be created at class level & shared by every object of that class.

Ex:
```
class Test
{
    int x=10;
    static int y=20;
    P S V m(-)
    {
        Test t1=new Test();
        t1.x=888;
        t1.y=999;
        Test t2=new Test();
        S.o.p(t2.x+"..."+t2.y);
    }
}
```

Op: 10...999



***

→ We can access static members directly from both instance & static areas, but we can't access instance members directly from static area.

→ But we can access instance members directly from instance area.

Q: Consider the 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);
}

⇒ Within a class which of the following declarations we can take simultaneously.

A) I & III
B) I & IV → CE: non-static variable x cannot be referenced from static context
C) II & III
D) II & IV
E) I & II → CE: x is already defined in Test
F) III & IV → CE: m1() is already defined in Test

***

Note:- instance & static variables with the same name is not allowed but instance & local variables or static & local variable can have same name.

abstract vs static:-

→ For static methods compulsory implementation should be available where as for abstract methods implementation should not be available.

Hence abstract static combination is illegal for methods.

Case(i):

→ Overloading concept is applicable for static methods including main(-) method also.
→ But JVM is always call String[] argument method only.
→ The other overloaded methods we have to call explicitly then it will be executed just like normal method call.

Ex:

```
class Test
{
    p s v main(String[] args)
    {
        S.o.p("String[]");
    }
    p s v main(int[] args)
    {
        S.o.p("int[]");
    }
}
```

(Overloaded methods)

o/p : String[]

Case(ii):

→ Inheritance concept is applicable for static methods including main(-) method.

→ Hence while executing child class if child class doesn't contain main(-) method then parent class main(-) method will be executed.

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

javac P.java ↵

P.class C.class

java P ↵

o/p : parent main

java C ↵

o/p : parent main

Case(iii):

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

It is method hiding but not overriding

javac P.java ↵

P.class C.class

java P ↵

o/p : parent main

java C ↵

o/p : child main

→ It seems overriding concept applicable for static methods, but it's not overriding and it is method hiding.

Note:- Inside a method, if we are using any instance variable compulsory we should declare that method as instance method.

Inside a method, if we are not using any instance variable then we should declare that method as static whether we are using static variables or not.

Ex:
```
class Student
{
    String name;
    int marks;
    int rollno;
    static String cname;

    public String getName()
    {
        return name;
    }

    public int getMarks()
    {
        return marks;
    }

    static String getCollegeName()
    {
        return cname;
    }

    getStudentInfo()
    {
        return name+" "+rollno+" "
               +cname;
    }

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

instance → getName(), getMarks(), getStudentInfo()
static → getCollegeName(), getAverage()

Native :—

→ native is the modifier applicable only for methods but not for classes and variables.

→ The methods which can be implemented in non-Java (mostly c and c++) are called native or foreign methods.

→ The main purposes of native keyword in Java are

1. To improve performance of the system.
2. To achieve machine level communication.
3. To use already existing legacy non-Java code etc.)

Pseudo code to use native keyword in Java:—

Ex:
```
class Native
{
    static
    {
        System.loadLibrary("native library path");
    }

    public native void m1();
}
```

① Load native libraries
② Declare native method

```
class Client
{
    p s v m(-)
    {
        Native n=new Native();
        n.m1();
    }
}
```

③ Invoke native method

→ For native methods implementation is already available & hence we are not responsible to provide implementation. Due to this native method declaration should ends with semicolon (;).

Eg: public native void m1();

✗ public native void m1(){} → CE: native methods cannot have a body

abstract Vs native:—

→ For native methods implementation is already available where as abstract methods implementation should not be available. Hence abstract native combination is illegal for methods.

native Vs strictfp:—

→ We can't declare native method as strictfp becoz there is no guarantee that native languages follow IEEE 754 standard. Hence native strictfp combination is illegal for methods.

→ For native methods the following concepts are applicable.

1. Inheritance
2. Overloading
3. Overriding

→ The main advantage of native keyword is performance will be improved, but the main disadvantage of native keyword is it breaks platform independent nature of Java.

Synchronized keyword:—

→ synchronized modifier applicable only for methods & blocks, but not for classes & variables.

→ If multiple threads operating simultaneously on same Java object then there may be a chance of data inconsistency problems.

→ To overcome this problem we should go for synchronized.

→ If a method or block declared as synchronized then at a time only one thread is allowed to operate on given method or block on the given object.

→ So that we can resolve data inconsistency problems.

→ The main advantage of synchronized keyword is we can overcome data inconsistency problems.

→ But the main disadvantage is it increases waiting time of threads and creates performance problems.

→ Hence if there is no specific requirement then it is never recommended to use synchronized keyword.

→ synchronized method should compulsory contain implementation where as abstract method should not contain implementation.

Hence abstract synchronized combination is illegal for methods.

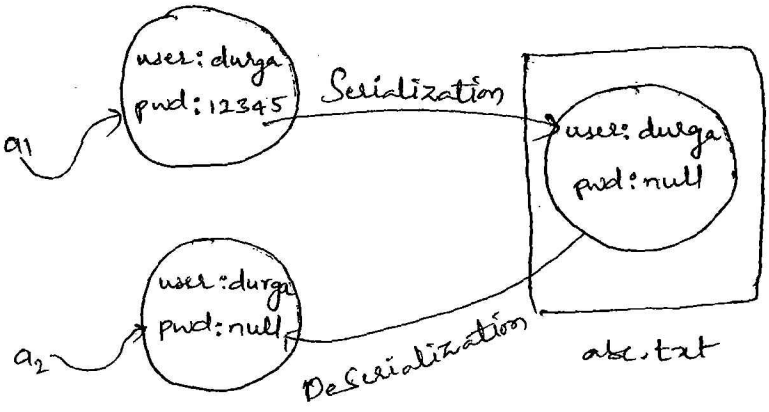
transient Keyword :—

→ transient modifier applicable only for variables, but not for methods & classes.

→ We can use transient keyword in Serialization.

→ While performing Serialization if we don't to save the value of a particular variable to meet security constraints. such type of variables we have to declare with transient keyword.

→ At the time of Serialization JVM ignores original value of transient variable & save default value to the file.



Volatile modifier:— is applicable

→ volatile modifier only for variables but not for methods & classes.

→ If the value of a variable keep on changing by multiple threads then there may be a chance of data inconsistency problems.

→ To overcome this problem we should go for volatile modifier.

→ If a variable declared as volatile then for every thread a separate copy will be created & all intermediate modifications performed by that thread will takes place in the local copy. So that there is no effect on remaining threads.

→ The main advantage of volatile keyword is we can overcome data inconsistency problems, but creating & maintaining a separate copy for every thread increases complexity & creates performance problems.

→ Hence volatile keyword is almost outdated & not recommended to use.

→ Volatile variable means its value keep on changing where as final variable means its value never changes.

*** Hence volatile final combination is illegal for variables.

Conclusions:—

→ The only applicable modifier for local variables is final.

*** → The modifiers which are applicable for constructors are public, private, protected and <default>.

→ The modifiers which are applicable for inner classes but not for outer classes are private, protected and static.

→ The modifiers which are applicable for classes but not for interfaces are final.

→ The modifiers which are applicable for classes but not for enum are final and abstract.

→ The modifiers which are applicable only for methods native.

→ The modifiers which are applicable only for variables transient & volatile.

Modifiers	class		methods	variables	blocks	interfaces		enum		constructor
	Outer	Inner				Outer	Inner	Outer	Inner	
public	✓	✓	✓	✓		✓	✓	✓	✓	✓
default	✓	✓	✓	✓		✓	✓	✓	✓	✓
private		✓	✓	✓			✓		✓	✓
protected		✓	✓	✓			✓		✓	✓
final	✓	✓	✓	✓		X	X	X	X	
abstract	✓	✓	✓	X		✓	✓	X	X	
static		✓	✓	✓	✓		✓		✓	
synchronized			✓	X	✓					
native			✓	X						
strictfp	✓	✓	✓	X		✓	✓	✓	✓	
transient			X	✓						
volatile			X	✓						

PLOT NO.56, HARSHA CHAMBERS, SAI NAGAR,NEAR BIG C MOBILES, MADHAPUR-500034, MOBILE:9505905786

4. Interfaces :—

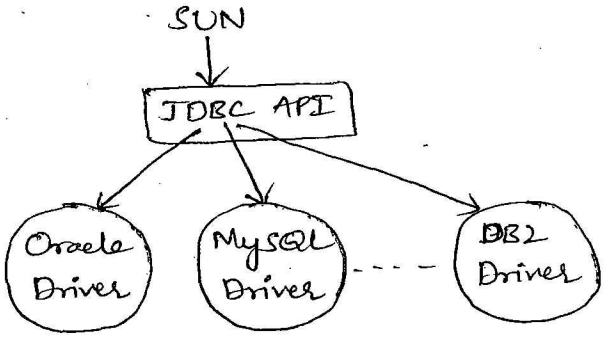
1. Introduction
2. interface declaration & implementation
3. extends Vs implements
4. interface methods
5. interface variables
6. interface naming conflicts
    1. Method naming conflicts
    2. variable naming conflicts
7. Marker interface
8. Adapter classes
9. interface vs abstract class vs concrete class
10. Differences btw interface & abstract class.

1. Introduction :—

Def^{n} ①: Any service requirement specification is considered as an interface.

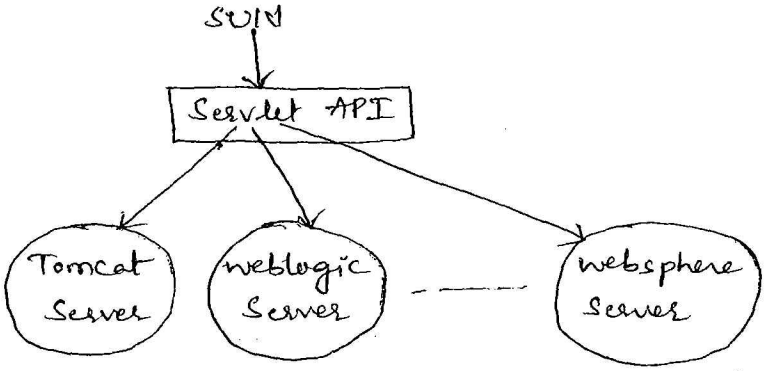
Ex ①: JDBC API acts as requirement specification to develop database Driver.

Database vendor is responsible to implement this specification.



Ex ②: Servlet API acts as requirement specification to develop webserver.

Webserver vendor is responsible to provide implementation.



Def^{n} ②: From the client point of view, interface defines the set of services what he is expecting.

From the service provider point of view, interface defines the set of services what he is offering.

Hence interface acts as contract blw client & service provider.

Ex: From the customer pointer of view ATM GUI Screen represents the set of services what he is expecting.

At the same time the GUI screen represents the set of services what bank people are offering.

Hence this GUI Screen acts as contract blw customer & bank.

Def^{n} ③: Inside interface every method is abstract. Hence interface is also considered as 100% pure abstract class.

***Summary Def^{n}s :-

→ Any service requirement specification (SRS)

(or)

Any contract blw client & service provider

(or)

100% pure abstract class is nothing but interface.

2. interface declaration and implementation :-

1. Whenever we are implementing an interface for each & every method of that interface we should provide implementation, o.w. we have to declare the class as abstract.

In this case, child class is responsible to provide implementation.

2. Whenever we are implementing interface methods compulsory we should declare that method as public, o.w. we will get CE.

Ex:
```
interface Interf
{
    void m1();
    void m2();
}
abstract class ServiceProvider implements Interf
{
    public void m1()
    { }
}
class SubServiceProvider implements ServiceProvider
{
    public void m2()
    { }
}
```

3. extends Vs implements :—

→ A class can extend only one class at a time where as an interface can extend any no. of interfaces simultaneously.

Ex:
```
interface A { }        interface B { }

interface C extends A, B
{ }
```
✓

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

→ A class can extend another class and can implement any no. of interfaces simultaneously.

Q: Which of the following is true?

1. A class can extend any no. of classes at a time. ✗
2. An interface can extend only one interface at a time. ✗
3. An interface can implement any no. of interfaces at a time. ✗

✗④. A class can implement only one interface at a time.

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

✓⑥. None of the above.

Q: Consider the following expression.

① . `X extends Y`

For which of the following possibilities the above expression is valid?

✗① Both X and Y should be classes

✗② Both X and Y should be interfaces

✓③ Both X and Y should be either classes or interfaces

④ No restrictions.

② `X extends Y, Z`

X, Y, Z should be interfaces

③ `X implements Y, Z`

X ⟶ class

Y, Z ⟶ interfaces

④ `X extends Y implements Z`

X and Y ⟶ classes

Z ⟶ interface

⑤ `X implements Y extends Z`

CE → becoz we have to take extends first followed by implements.

4. interface methods :—

→ Every interface method is always public & abstract whether we are declaring or not.

Ex:
```
interface Interf
{
    void m1();
}
```

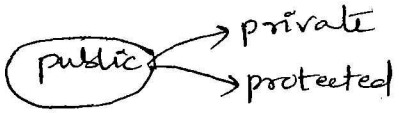
→ public : To make this method available to every implementation class.

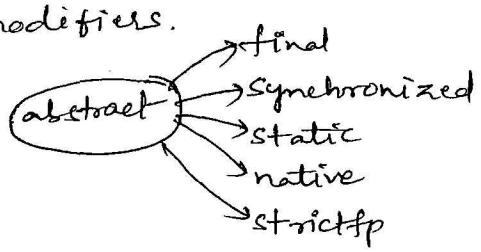
→ abstract : Implementation class is responsible to implement this method.

→ Hence the following method declarations are equal inside interface.

Ex:
```
void m1();
public void m1();
abstract void m1();
public abstract void m1();
```
equal

→ As every interface method is always public & abstract whether we are declaring or not, hence we can't declare interface methods with the following modifiers.





Q: Which of the following method declarations are valid inside interface?

✗ ①. public void m1(){}

✗ ②. private void m1();

✗ ③. protected void m1();

✗ ④. static void m1();

✗ ⑤. public abstract native void m1();

✓ ⑥. abstract public void m1();

5. Interface variables :—

→ An interface can contain variables.

→ The main purpose of interface variables is to define requirement level constants.

→ Every interface variable is always public static final whether we are declaring or not.

Ex: interface Interf
{
int x=10;
}

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

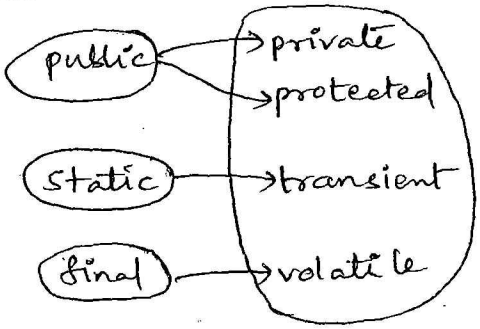
→ static : Without existing object also, implementation class has to access this variable.

→ final : Implementation class can access this variable but can't modify becoz it is common variable for several implementation classes.

→ Hence the following variable declarations inside interface are equal.

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 with the following modifiers.



*** → For the interface variables compulsory we should perform initialization at the time of declaration only, o.w. we will get CE.

Ex: interface Interf
{
int x; → CE: = expected
}

→ we can access interface variables inside implementation class but we can't modify.

Ex:
```
interface Interf
{
    int x=10;
}
```

```
class Test implements Interf
{
    p s v m(-)
    {
        x=999;
        S.o.p(x);
    }
}
```
CE: cannot assign a value to final variable x

```
class Test implements Interf
{
    p s v m(-)
    {
        int x=999;
        S.o.p(x);
    }
}
```
⇒ O/P : 999

Q: Which of the following variable declarations are valid?

✗ ①. int x;
✗ ②. private int x=10;
✗ ③. protected int x=10;
✗ ④. volatile int x=10;
✗ ⑤. transient int x=10;
✓ ⑥. public static int x=10;

6. interface naming conflicts :—

*1 Method Naming Conflicts :—

Case (i): If two interfaces contain a method with same signature and same return type then in the implementation class one method implementation is enough.

Ex:
```
interface Left
{
  public void m1();
}

interface Right
{
  public void m1();
}

class Test implements Left, Right
{
  public void m1()
  {
  }
}
```

Case (ii): If two interfaces contain a method with same name but with different arguments then in the implementation class we have to provide implementation for both methods and these methods acts as overloaded methods.

Ex:
```
interface Left
{
  public void m1();
}

interface Right
{
  public void m1(int i);
}

class Test implements Left, Right
{
  public void m1()
  { }
  public void m1(int i)
  { }
}
```
(overloaded methods)

Case(iii): If two interfaces contain a method with same signature but different return types then it is impossible to implement both interfaces simultaneously.

Ex:
```
interface Left
{
  public void m1();
}
```

```
interface Right
{
  public int m1();
}
```

→ We can't write any Java class which implement both interfaces simultaneously.

*** Q: Is a Java class can implement any no. of interfaces simultaneously?

Ans: Yes, except if two interfaces contain a method with same signature but with different return types.

2. Interface variable naming conflicts:-

Case(i): If two interfaces can contain a variable with same name & there may be a chance of variable naming conflict, but we can resolve by using interface names.

Ex:
```
interface Left
{
  int x=888;
}
```

```
interface Right
{
  int x=999;
}
```

```
class Test implements Left, Right
{
  p s v m(-)
  {
    S.o.p(x); → CE: reference to x is ambiguous
    S.o.p(Left.x); ⇒ o/p : 888
    S.o.p(Right.x); ⇒ o/p : 999
  }
}
```

8. Adapter classes:-

→ An Adapter class is a simple Java class that implements an interface with only empty implementations.

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

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

→ If we implement an interface directly compulsory we should provide implementation for each & every method of that interface. Whether we are interested or not.

Ex:
```
class Test implements X
{
    m3()
    {
    }   ≡ 10 lines
    m1(){}
    m2(){}
    m4(){}
     :
    m1000(){}
}
```
(1000 lines)

→ The problem in this approach is it increases length of the code and reduces readability, it increases complexity of the programming.

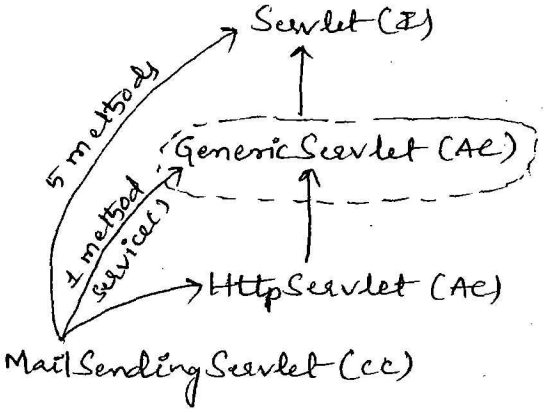
→ But we can resolve this problem by using Adapter class.

→ Instead of implementing interface if we extend Adapter class then we have to provide implementation only for required methods, but not for total methods of interface.

Ex:
```
class Test extends AdapterX
{
    m3()
    {
    }
}
```
```
class Demo extends AdapterX
{
    m7()
    {
    }
}
```

→ The advantage of this approach is length of the code will be reduced & readability will be improved.

Ex: We can develop a servlet either by implementing Servlet interface or by extending GenericServlet.



If we implement Servlet interface directly then compulsory we should provide implementation for all 5 methods of Servlet interface whether it is required or not.

It increases length of the code & reduces readability.

Instead of implementing Servlet interface directly if we extend GenericServlet then we have to provide implementation only for required method service() and we are not responsible to implement all Servlet interface methods.

Hence more or less GenericServlet acts as Adapter class for Servlet interface.

*** 7. Marker interface :—

→ An interface which doesn't contain any methods and by implementing that interface if our objects will get some ability such type of interfaces are called Marker interfaces.

→ Marked interfaces are also called Ability (or) Tagged interfaces.

Ex: Serializable
Cloneable
RandomAccess
SingleThredModel
} All these interfaces are marked for some ability.

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

Ex②: By implementing Cloneable interface our objects can able to produce exactly duplicate cloned objects.

***Q: Without having any methods how we are getting ability in marker interfaces?

Ans: Internally JVM is responsible to provide required ability.

***Q: Why JVM is providing required ability in marker interfaces?

Ans: To reduce complexity of the programming.

***Q: Is it possible to define our own marker interfaces?

Ans: Yes, we can define our own marker interfaces but customization of JVM must be required.

Ex: Sleepable, Jumpable, ----

***9. 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.

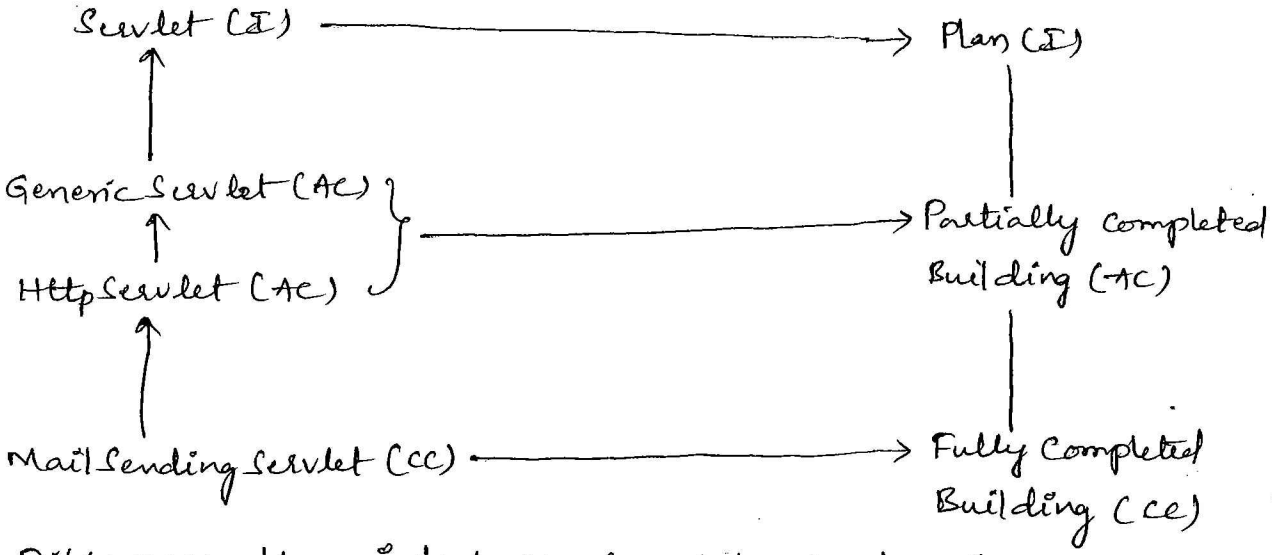
Ex: Servlet(I)

→ If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.

Ex: GenericServlet(AC)

→ If we are talking about implementation completely & ready to provide service then we should go for concrete class.

Ex: MailSendingServlet(CC)



*** 10. Differences b/w interface & abstract class :—

interface	abstract class
1. If we don't know anything about implementation just we have requirement specification then we should go for interface.	1. If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.
2. Every method present inside interface is always public & abstract whether we are declaring or not.	2. Every method present inside abstract class need not be public & abstract, we can take concrete methods also.
3. We can't declare interface methods with the following modifiers private, protected, static, final, synchronized, native & strictfp.	3. There are no restrictions on abstract class method modifiers.
4. Every variable present inside interface is always public static final whether we are declaring or not.	4. Every variable present inside abstract class need not be public static final.

interface	abstract class
5. We can't declare interface variables with the following modifiers private, protected, transient & volatile.	5. There are no restrictions on abstract class variable modifiers.
6. For interface variables compulsory we should perform initialization at the time of declaration, o.w. we will get CE.	6. For abstract class variables which is not required to perform at the time of declaration.
7. Inside interface we can't declare static & instance blocks.	7. Inside abstract class we can declare instance & static blocks.
8. Inside interface we can't take constructor.	8. Inside abstract class we can take constructor.

*** Q: We can't create object for abstract class but abstract class can contain constructor, what is the need?

Ans: abstract class constructor will be executed to perform initialization of child object at the time of child object creation.

*** Note :- ① Either directly or indirectly we can't create object for abstract class.

② whenever we are creating child class object parent constructor will be executed, but parent object won't be created.

*** Q: interface contains only abstract methods, but abstract class also can contain only abstract methods then what is the need of interface?

(Q)

Is it possible to replace interface concept with abstract class?

Ans: We can replace interface with abstract class, but it is not a good programming practice (which is like recruiting IAS officer for sweeping purpose).

Approach ①:

```
interface X
{ ≡ }

class Test implements X
{
  ≡
}
```

① Test class can extend some other class while implementing X.

② Object creation is not costly.

```
Test t=new Test();
```

2 min

Approach ②:

```
abstract class X
{ ≡ }
class Test extends X
{
  ≡
}
```

① Test class can't extend any other class while extending X

② Object creation is costly.

```
Test t=new Test();
```

10 min

→ If everything is abstract then it is highly recommended to go for interface.