Declarations & Access Modifiers

① Java Source file Structure (1-9) Packages - 2

② Class modifiers (10-14)

③ member modifiers (15-23)

*④ Interfaces. (24-31)

Java Source file Structure :-

→ A Java program can Contain any no. of classes but atmost one class can be declared as the public. if there is a public class the name of the program & name of public class must be matched otherwise we will get CompiletimeError.

→ If there is no public class then we can use any name as Java Source file name, there are no restrictions.

Ex:-
```
Class A
{
}
Class B
{
}
Class C
{
}
```

Save: Sai.java ✓
R.java ✓
D.java ✓

Case(1):-

If there is no public class then we can use any name as java source file name.

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

Case2 :→

If class B declared as public & the program name is A.java Then we will get CompiletimeError Saying,

" Class B is public should be declared in a file named B.java"

Case 3 :-

If we declare Bothe A & B classes as public & name of the program is B.java then we will get CompiletimeError Saying.

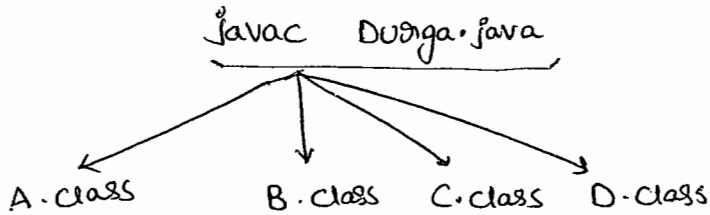
" Class A is public should be declared in a file named A.java".

Ex:-

```
Class A
{
  P.S.V.m(String[] args)
   {
     S.o.pln("A class main method");
   }
}
Class B
{
  P.S.V.m(String[] args)
   {
     S.o.pln("B class main method");
}}
```

```
Class C
{
  P.S.V.m(String[] args)
   {
     S.o.pln("C class main method");
   }
}
Class D
{
}
```

Save ⇒ Durga.java



① Java A ↵

A class main method

② Java B ↵

B class main method

③ Java C ↵

C class main method

④ Java D ↵

R.E:- NoSuchMethodError : main

⑤ Java Durga ↵

R.E:- NoClassDefFoundError : Durga

Note:-

→ It is highly recommended to take only one class per source file & name of the file and that class name must be matched. This approach improves readability of the code.

Import Statement :-

```
Class Test
{
 P.S.V.m(String[] args)
 {
  ArrayList l = new ArrayList(); // ArrayList()
 }
}
```

C.E:- symbol: method ArrayList

C.E:- Cannot find Symbol
Symbol: class ArrayList
Location: class Test

→ We can resolve this problem by using fully Qualified name
java.util.

→ the problem with usage of fully Qualified name everytime increases length of the Code & reduces readability.

→ we can resolve this problem by using import Statement

```
import java.util.ArrayList;
Class Test {
 P.S.V.m(String[] args)
 {
  AL l = new AL();
 }
}
```

→ Where ever we are using import statement it is not required to use fully Qualifiedname hence it reduces improves readability & reduces Length of the Code.

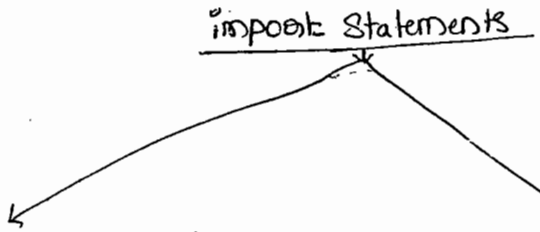
Case(1):-

Types of import Statements:-

↳ There are 2 types of import Statements

(1) Explicit class import

(2) Implicit class import



Explicit class import :-

Ex!. import java.util.AL;

→ This type of import is highly recommended to use. because it improves readability of the code.

→ Best Suitable for Hietch City Where readability is important

Implicit class import :-

Ex!- import java.util.*;

→ It is never recommended to use this type of import because it reduces readability of the code.

→ Best Suitable for Annexpet where typing is important.

Case 2!. difference blw #include & import Statement :-

→ In c language #include all the specified header files will be loaded at the time of include statement only irrespective of wheather we are using those header files are not. Hence this is static loading.

→ But in the case of java language import statement no .class file will loaded at the time of import statement, in the next lines of code when ever we are using a class at that time only the coorresponding .class file will be loaded, this type of loading is called dynamic loading or load on demand or load on fly.

Case 3!

Which of the following import statements are valid?

✗ ① import java.util;

✗ ② import java.util.AL.*;

✓ ③ import java.util.*;

✓ ④ import java.util.AL;

Case 4!.

→ Consider the code, class MyRemoteObject extends java.rmi.UniCastRemoteObject

{

=

}

→ the code compiles fine eventhough we are not using import statement because we used fully Qualified Name.

Note!.

→ when ever using fully Qualified name it is not required to use import statement. When ever we are using import statement it is not

Required to use fully Qualified name.

Case 5:

Example :-

Date, List } available in both util & sql

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

C.E :- "Reference to Date is ambiguous"

Note :-

even in List Case also we will get the same ambiguity problem because it is available in both util & sql packages.

Case 6 :-

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

order :-

1. Explicit class import
2. Classes present in Current Working directory
3. implicit class import.

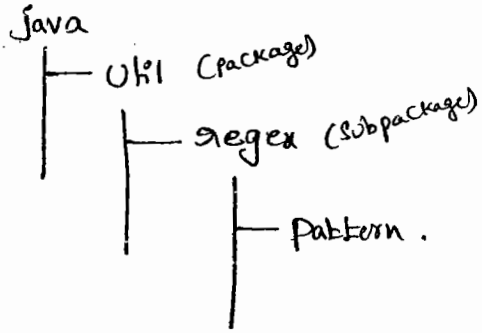
Conclusion :- While Resolving class names Compiler will always gives the precedence in the following order,

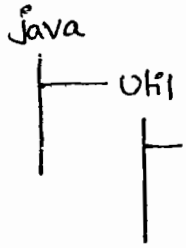
→ order see above

Case 7 :-

→ When ever we are importing a package all classes & interfaces present in that package are available, but not subpackage classes.

Ex :-





→ To use pattern class which of the following import is required

✗ ① import java.*;

✗ ② import java.util.*;

✓ ③ import java.util.regex.*;

✓ ④ import java.util.regex.pattern;

Case (8) :-

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

① java.lang package.

② ~~java~~ default package (current working directory).

Case 9 :-

→ import statement is totally compiletime issue if no. of imports increases then compiletime will be increased automatically. but there is no effect on execution time.

Static import :-

→ This Concept introduced in 1.5 Version.

→ According to SUN Static import improves readability of the Code, But according to World wide programming Experts (Like us) Static imports reduces the readability of the Code & creates Confusion, it is not recommended to use Static import if there is no Specific requirement

→ Usually we can access Static members by using class names, but when ever we are using Static import, it is not required to use Class name and we can access Static members directly.

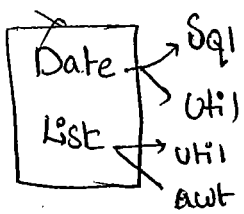
ex:-

Without Static import

```
class Test
{
 p.s.v.m(String[] args)
 {
  S.o.pln(Math.sqrt(4));
  S.o.pln(Math.random());
  S.o.pln(Math.max(10,20));
 }
}
```

With Static import

```
import static java.lang.math.sqrt;
import static java.lang.maths.*;
class Test
{
 p.s.v.m(String[] args)
 {
  S.o.pln(sqrt(4));
  S.o.pln(random());
  S.o.pln(max(10,20));
 }
}
```



* Explain about System.out.println() :-

```
Class Test
{
  p.static String name = "xyz";
}

Test.name.length();
```

- Test → It is a class name
- name → Static variable present in Test class of the type String
- length() → It is a method present in String class

```
Class System
{
  Static PrintStream out;
}

System.out.println();
```

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

Explanation :-

→ Out is a static variable present in System class hence we can access by using classname.

→ But when ever we are using static import it is not required to use class name we can access out variable directly.

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

-ve prospective (Ambiguity):-

Ex:-
```
import static java.lang.Integer.*;
import static java.lang.Byte.*;
Class Test
{
  p.s.v.m(String[] args)
  {
    S.o.pln(MAX_VALUE);
  }
}
```

C.E:- Reference to MAX_VALUE is ambiguity

Note:-
Two classes Contains a variable or method with Same name is Very Common Hence ambiguity problem is also Very Common in Static import.

Ex:-
→ While resolveing static members Compiler will always gives the precedence in the following order.

1. Current class static members
2. Explicit Static import
3. implicit Static import.

Ex:-

```
import static java.lang.Integer.MAX_VALUE;  ----> (2)
import static java.lang.Byte.*;  ----> (3)
Class Test
{
  static int MAX_VALUE =999;  ----> (1)
  P.S.V.m(String[] args)
  {
    S.O.pln(MAX_VALUE);
  }
}
```

→ If we are Commenting Line(1) then Explicit static import will get priority Hence we will get Integer class MAX_VALUE is o/p 2147483647.

→ If we are Commenting Lines(1) & (3) then Byte class MAX_VALUE will be Considered & we will get 127 as o/p.

(-ve point):-

→ Strictly Speaking usage of Class Name to access static variables & methods improves readability of the Code. Hence it is not recommended to use static imports.

Q) Which of the following import Statements are valid.

✗ ① import java.lang.math.*; (we should not use * after the class).

✗ ② import java.lang.math.Sqrt.*; (we should not use * after the method).

✗ ③ import static java.lang.math;

✓ ④ import java.lang.math;

✓ ⑤ import static java.lang.math.*;

✗ ⑥ import static java.lang.math.Sqrt(); → problems

✓ ⑦ import static java.lang.math.Sqrt;

Normal 'import' Vs Static import:-

→ We can use normal import to import classes & interfaces of a package. whenever we are using general import it is not required to use fully Qualified Name & we can use short names directly.

→ We can use Static import to import Static variables & methods of a class. when ever we are using Static import then it is not required to class name to access Static member we can access directly.

Packages

9348 classes are their in Java according to 1.6v

Package:-

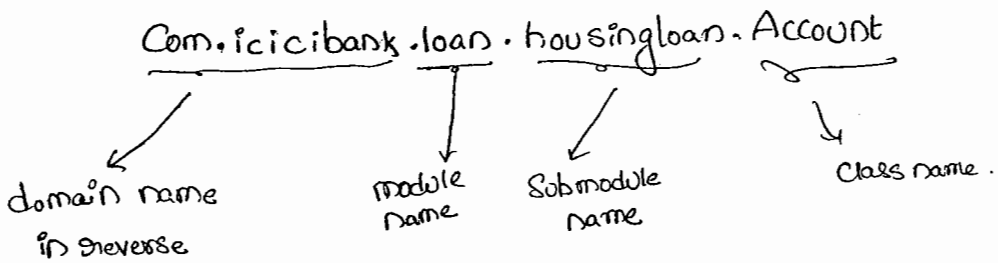
→ It is an Encapsulation mechanism to group related classes and interfaces into a single module. The main purposes of packages are

① To resolve naming conflicts.

② To provide Security to the classes & interfaces. So that outside person can't access directly

③ It improves modularity of the application.

→ There is one Universally accepted Convension to name packages i.e to use internet domain name in reverse.

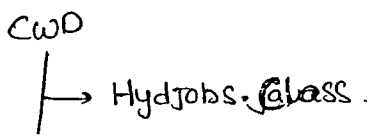


Ex:-

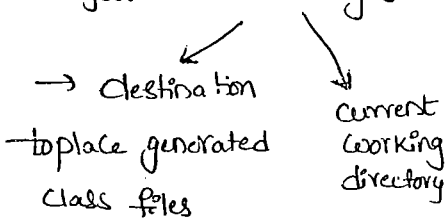
```
Package Com.durgajobs.itjobs;

Public class HydJobs
{
 P.S.V.m(String[] args)
 {
  S.o.pln("Getting Jobs is very easy");
 }
}
```

① javac HydJobs.java

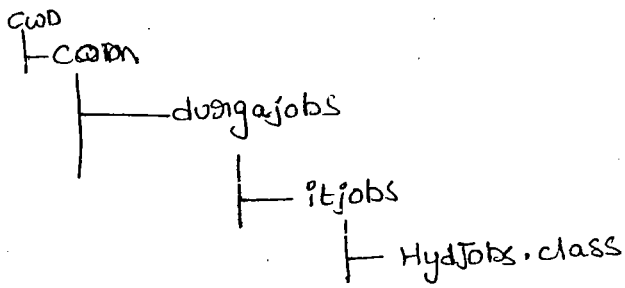
→ The generated class file will be placed in Current working directory



② javac -d . HydJobs.java



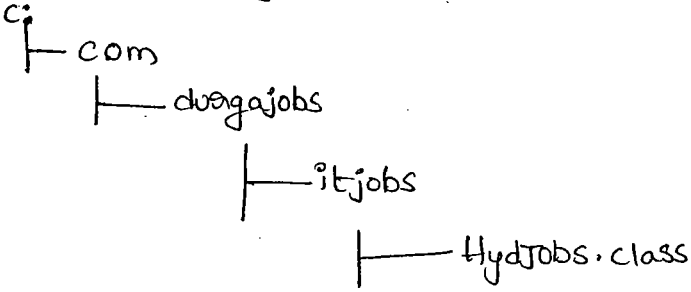
→ generated class file will be placed into corresponding package Structure.



→ If the specified package structure is not already available then this command itself will create that package structure.

→ As the destination we can use any valid directory

Ex: javac -d c: HydJobs.java



→ If the specified destination is not already available then we will get Compile time Error

Ex: javac -d z: HydJobs.java

→ If z: is not already available then we will get Compiletime Error.

Run

→ Java com.durgajobs.itjobs.HydJobs ←

o/p :- Getting job is very easy.

Conclusions:-

① → In any java program there should be only atmost 1 package statement. If we are taking morethan one package statement we will get compiletime error.

Ex:- ✓ package pack1;

→ package pack2; ←

Class A

{

}

C.E:- class, interface or enum expected.

② In any java program the first non comment statement should be package statement (if it is available).

Ex:- ✓ import java.util.*;

→ package pack1;

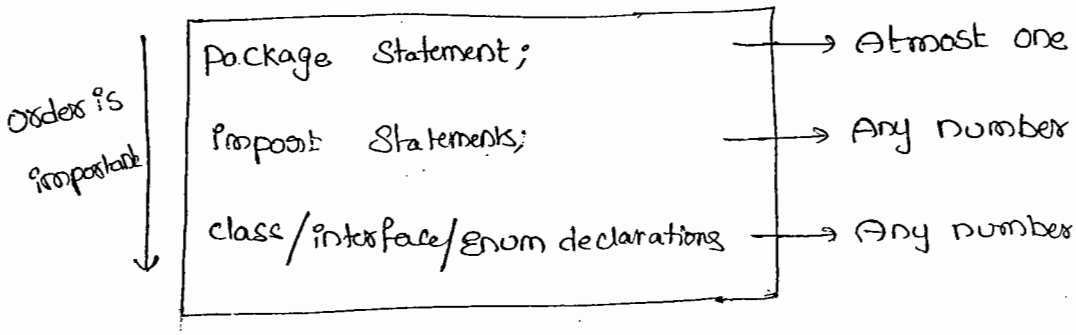
Class A

{

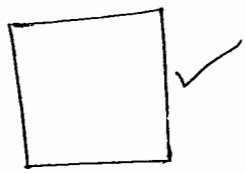
}

C.E:- class, interface or enum Expected.

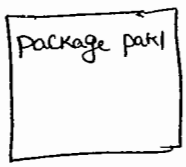
→ The proper Structure of a Java Source file is



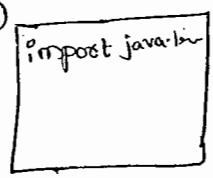
→ The following are Valid Java programs.

① 

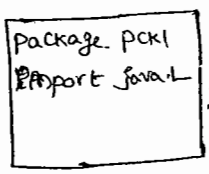
Test.java

② 

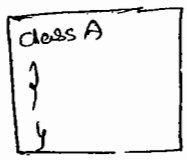
Test.java

③ 

Test.java

④ 

Test.java

⑤ 

Test.java

→ in this prog only creates .class file.

→ An Empty Source file is a Valid Java program.

** Class modifiers **

→ when ever we are coding our own java class compulsary we have to provide some information about our class to the JVM Like,

(1) Wheather our class can be accessable from any where or not

(2) Wheather child class creation is possible for our class or not.

(3) Wheather instanciation is possible or not e.t.c.

→ We can Specify this information by declaring with appropriate modifier.

→ The only applicable modifiers for top-level classes are

1) public
2) <default>
3) final
4) abstract
5) Strictfp.

→ If we are using any other modifier we will get CompiletimeError. Saying "modifier xxxxxxx not allowed here".

Ex:-
```
private class Test
{
  p.s.v.m(——)
  {
    int x=0;
    for(int y=0; y<3; y++)
    {
      x=x+y;
    }
    S.o.pln(x);
  }
}
```

C.E:- modifier private not allowed here

→ But for the Inner classes the following modifiers are allowed

(1) public
(2) <default>
(3) final
(4) abstract
(5) strictfp
(6) private
(7) protected
(8) static.

```
11: private class A
12: {
13: }
14: static class B
15: {
 :  }
 :  P.S.V.M.C ⟶
 :  {
        S.o.pln("Hi");
    }
```

because the main class is not declare

28/04/11

access Specifiers Vs access modifiers :-

→ In old languages like C & C++ public, private, protected & default are considered as access Specifiers.. & all the remaining like final, static are considered as access modifiers.

→ But in Java there is no such type of division all are considered as access modifiers.

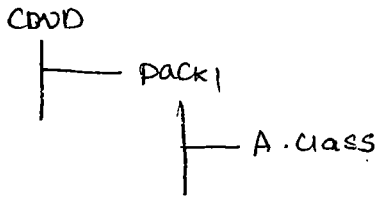
Public classes :-

→ If a Class declared as the public then we can access that class from any where.

Ex:-

```
Package pack1;
Public class A
{
 Public void m1()
 {
  S.o.pln("Hello");
 }
}
```

Javac -d . A.java



```
Package pack2;
import pack1.A;
Class B
{
 P.S.V.m(String[] args)
  {
     A a = new A();
     a.m1();
  }
}
```

Compl. javac -d . B.java ↵

Run! java pack2.B ↵

→ If we are not declaring Class A as public, then we will get Compile-time Error while Compiling B class, Saying "pack1.A is not public in pack1; Can't be accessed from outside package"

default classes :-

→ If a class declared as default then we can access that class only with in that current package. i.e from outside of the package we can't access.

final modifier :-

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

→ If a method declared as the final then we are not allowed to override that method in the child class.

Ex!-

```
Class P
{
  public void property()
  {
    S.o.pln("money + Goold + Land");
  }
  Public final void marry()
  {
    S.o.pln("subba laxmi");
  }
}
Class C extends P
{
  public void marry()
  {
    S.o.pln("kajal | asba | atara");
  }
}
```

C.E

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

→ If a class declared as the final then we can't create child class

Ex!-

```
final Class P
{
}
```

```
Class C extends P
{
}
```

X

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

C.E:- Can't inherit from final P.

↳ Every method present inside a final class is always final bydefault. but every variable present in final class need not be final.

→ The main Advantage of final keyword is we can acheive Security as no one is allowed to change our implementation.

→ But the main disAdvantage of final keyword is we are missing key benefits of OOP's Inheritance & polymorphism (overriding). Hence, if there is no specific requirement never recommended to use final keyword.

**abstract modifier :-

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

abstract method :-

→ Eventhough we dont ~~about~~ know about implementation Still we can declare a method with abstract modifier. i.e abstract methods can have only declaration but not implementation. Hence, Every abstract method declaration should Compulsary ends with ";".

Ex:-

X 1) public abstract void m1(){}

✓ 2) public abstract void m2();

→ Child classes are responsible to provide implementation for parent class abstract methods.

Ex:-

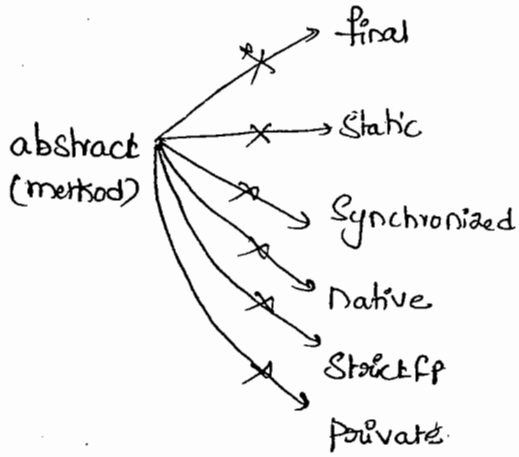
```
abstract class Vechicle
{
  public abstract int getNoofWheels();
}

Class Bus extends Vehicle
{
  public int getNoOfWheeels()
  {
    Return 6;
  }
}

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

→ By declaring abstract methods in parent class we can define Guidelines to the child classes which describes the methods those are to be Compulsary implemented by child class.

29/04/11

→ abstract modifier never talks about implementation, if any modifier talks about implementation then it is always illegal combination with abstract.

→ The following are various illegal combinations of modifiers for methods



abstract class :-

→ for any java class if we don't want instanciation then we have to declare that class as abstract. i.e, for abstract classes instanciation (creation of object) is not possible.

Ex:-
```
abstract class Test
{
}
Test t = new Test();
```

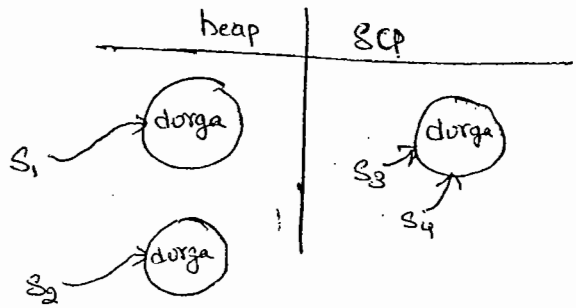
C.E:- Test is abstract; cannot be instantiated
Test t = new Test();

Ex:- String S_1 = new String("durga");

String S_2 = new String("durga");

String S_3 = "durga";

String S_4 = "durga";



otm Sai Ramprabu.

1) notify() & notifyall()
2) Collection & Collections
3) equals() & ==
4) Comparable & Comparator
5) String & StringBuffer
6) StringBuffer & StringBuilder
7) throw & throws
8) throws & thrown
9) HashMap & Hashtable
10) enum, Enum, Enumeration
11) final, finally, finalizer

durga Sir.

1) Language fundamentals
2) Operators & Assignments
3) Flow-Control
4) declaration & Access modifier
5) oops concepts
6) Exception Handling
7) multi threading
8) Inner classes
9) java.lang package
10) java.io package
11) Serialization
12) java.util.package (Collection framework)
13) Generics
14) Regular Expressions
15) G.C
16) Assertions (1.4)
17) JDBC
18) enum
19) developement

Dell

chaitanyaobjs@gmail.com
Satish4dworld@
29444524
Chait
Chaithanya_Anumanchi@dell.com.

ON durgajobsInfo 9870807070

abstract class Vs abstract method :-

→ If a class Contains atleast one abstract method then Compulsary That class should be declared as abstract otherwise we will get CompileTimeError. because, the implementation is not Complete & hence we Can't Create an object.

→ Eventhough Eas class doesnot Contain any abstract method still we Can declare the class as abstract. i.e, abstract class Can Contain Zero "0 no of abstract method.

Ex:- HTTPServelet, this class doesn't Contain any abstract method but still it is declared as abstract.

Ex:-

① Class Test
{
public void m1();
}
C.E:- missing method body, or declare abstract

② class Test
{
public abstract void m1()
{
}
}
C.E:- abstract methods Can't have a body

③ Class Test
{
public abstract void m1();
}

C.E:- Test is not abstract and doesn't override abstract method m1() in Test.

Ex-4:-

```
abstract class Test
{
  public abstract void m1();
  public abstract void m2();
}
class SubTest extends Test
{
   public abstract void m1(){}
}
```

C.E:- SubTest is not abstract and does not override abstract method m2() in Test

→ We can handle these Compiletime Errors either by declaring SubTest as abstract class or by providing implementation for m2().

Note:-

→ The usage of abstract methods, abstract class & interfaces are recommended & it is always good programing practice.

abstract Vs final :-

→ abstract methods we have to override in child classes to provide implementation. where as final methods can't be overridden. Hence, abstract final combination is illegal combination for methods.

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

→ final class Can't have abstract methods where as abstract class Can Contain final methods.

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

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

Strictfp (all lowercase) modifier :- (strictfloatingpoint)

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

→ if a method declared as strictfp all floatingpoint caluclations in that method has to follow IEEE 754 standard so, that we will get platform independent results.

→ strictfp method always talks about implementation where as abstract method never talks about implementation. Hence strictfp-abstract method Combination is illegal Combination for methods.

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

→ abstract – strictfp Combination is legal for classes but illegal for methods

Ex:- abstract strictfp class Test
{
}
✓

public abstract strictfp void m1(); ✗ (invalid).

Membea (variables & methods) modifiers :-

① public members :-

→ If we declare a member as public then we can access that member from anywhere but coorresponding class should be visible (public) i.e, Before checking member visibility we have to check class visibility.

Ex:-

```
Package pack1;

Class A
{
  public void m1()
  {
    S.o.pln("hi");
  }
}
```

```
Package pack2;
import pack1.A;
Class B
{
 p.s.v.m(——>)
 {
   A a = new A();
   a.m1();
 }
}
```

✗

→ Eventhough m1() method is public, we can't access m1() from outside of pack1 because the coorresponding class A is not declared as public. if both are public then only we can access.

② default members :-

→ If a member declared as the default, then we can access that member only with in the current package & we can't access from outside of the package. 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 with in the current class.

→ abstract methods should be visible in child classes to provide implementation where as private methods are not visible in child classes. Hence private-abstract combination is illegal for methods.

④ protected members : (The most misunderstood modifier in java) :-

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

protected ≡ <default> + kids (on another package only child reference).

→ with in the current package we can access protected members either by parent reference or by child reference.

→ But from outside package we can access protected members only by using child reference. if we are trying to use parent reference we will get C.E

Eg:.

```
package pack1;
public class A
{
  protected void m1()
  {
    S.o.pln("the most misunderstood modifier in Java");
  }
}
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();

Package pack2;

import pack1.A;

public 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();

}

}

→ The most restricted modifier is "private"

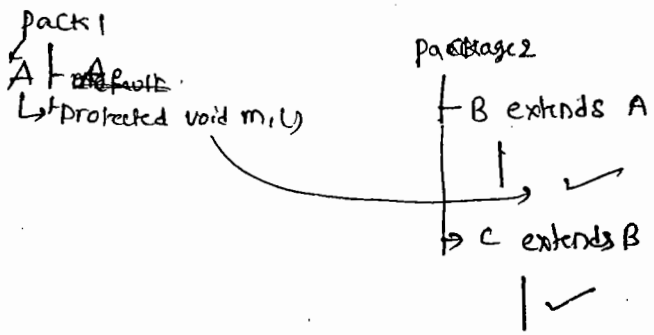
→ The most accessible modifier is "public"

→

private < default < protected < public

→ The recommended modifier for variables is private

→ The recommended modifier for methods is public



→ The most restricted

* private < default < protected < public

Visibility	private	<default>	protected	public
① With in 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.	×	×	✓ (But we should use only child class reference)	✓
⑤ from non-child class of outside package.	×	×	×	✓

"final" variables :-

→ In General for instance & static variables it is not required to perform initialization Explicitly JVM will always provide default values.

→ But for the local variables JVM won't to provide any default values Compulsory we should provide initialization before using that variable.

"final instance variables" :-

→ for the normal instance variables it is not required to perform initialization Explicitly JVM will provide default values.

→ If the instance variable declared as the final then Compulsory we should perform initialization wheather we are using or not otherwise

we will get Compiletime Error.

ex!-

```
Class Test
{
  int x;
}
```

✓

```
Class Test
{
  final int x;
}
```

✗

C.E!- variable x might have not been initialized.

Rule:-

(***) for the final instance variables we should perform initialization before Constructor Complition.

→ i.e, the following are various places for this,

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;
  }
}
```

✓

→ Other than these if we are perform initialization any where else we will get Compiletime Error.

Ex:-

```
Class Test
{
    final int x;
    public void m1()
    {
        x=10;   X   C.E:- Cannot assign a value to final variable x.
    }
}
```

final Static Variables:-

→ for the normal Static variables it is not required to perform initialization Explicitly, JVM will always provide default values.

→ But for final static variables we should perform initialization Explicitly Otherwise we will get C.E.

Ex:-

```
Class Test
{
    Static int x;
}
```
✓

```
Class Test
{
    final Static int x;
}
```
X

C.E:- Variable x might not have been initialized.

Rule:-

* for the final Static variables we should perform initialization Before Class loading Complition.
* i.e, the following are various places to perform this

(1) At the time of declaration

eg:- class Test

{

final static int x=10;

}

(2) Inside Static Block

eg:- class Test

{

final static int x;

static

{

x=10;

}

}

→ If we are performing initialization any where else we will get Compiletime Error.

class Test

{

final static int x;

public void m1()

{

x=10;

}

}

C.E:- Can't assign a value to final variable x.

iii) final Local variables :-

→ for the local variables JVM won't to provide any default values Compulsory we should perform initialization before using that variable.

Eg:- ①
```
class Test
{
    public static void main()
    {
        int x;
        S.o.pln("Hello");
    }
}
```
o/p :- Hello

②
```
class Test
{
    public static void main()
    {
        int x;
        S.o.pln(x);
    }
}
```
C.E:- variable x might not have been initialized.

* → Eventhough Local variable declared as the final it is not required to perform initialization if we are not using that variable.

Eg:-
```
class Test
{
    P.S.V.M()
    {
        final int x;
        S.O.pln("Hello Sai");
    }
}
```
o/p :- Hello Sai.

*** → the only applicable modifier for local variables is final. if we are using any other modifier we will get Compiletime Error.

ex:-
```
class Test {
    P.S.V.M() {
        public int x=10;
        private int x=20;
        static int x=60;
        protected int x=30;
        final int x=40;
```

→ formal parameters of a method simply access as local variables of that method. hence, a formal parameter can be declared as final.

→ If we declare a formal parameter as final within the method we can't change its value otherwise we will get compiletime Error.

Ex:-

```
Class Test
{
  p.s.v.m(---->)
  {
    m1(10,20);
  }
  p.s.v.m1(final int x, int y)
  {
    x=1000; // Can't assign a value to final variable x.
    y=2000;
    S.o.pln(x+"----"+y);
  }
}
```

Static → class level
instance → object level

Static modifier :-

→ Static is the modifier applicable for variables & methods but not for classes (but innerclass can be declared as static).

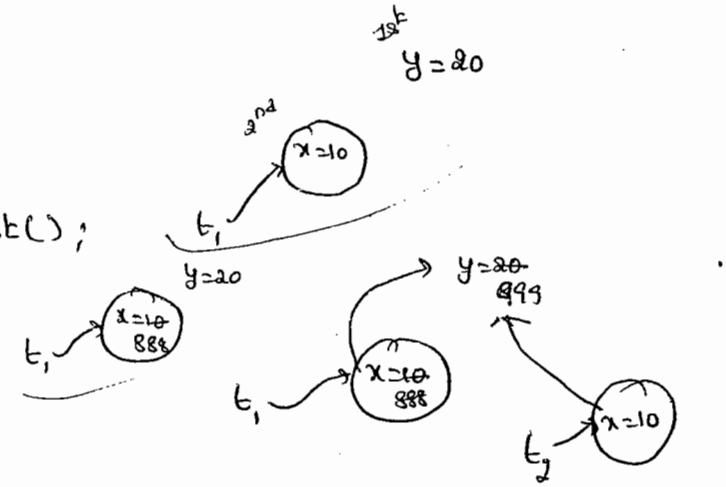
→ if the value of a variable is varied from object to object then we should go for instance variable. In the case of instance variable for every object a seperate copy will be created.

→ If the value of a variable is same for all objects then we should go for static variables. In the case of static variable only one copy will be created at class level and share that copy for every object of that class.

The beginning first Static variable is created at when class is created. 19

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.pln(t2.x + "---" + t2.y);
  }
}
```



for every object a Seperate copy will be Created.

10 999

→ Static members can be accessed from both instance & Static areas where as instance members can be accessed only from instance area directly. i.e, from static area we can't access instance members directly otherwise we will get Compiletime Error.

Q) Consider the following declarations

I. int x=10;

II. Static int x=10;

III. public void m1()
{
S.o.pln(x);
}

IV. public Static void m1()
{
S.o.pln(x);
}

→ which of the above we can take simultaneously with in the same class.

✓ A). I & III

✗ B). I & IV . CE !- non-static variable x cannot be accessed from static context

✓ C) II & III

✓ D) II & IV

✗ E) I & II

✗ F) III & IV

→ for static methods compulsary implementation should be available where as for abstract methods implementation should not be available Hence abstract-static combination is illegal for methods.

→ for static methods overloading concept is applicable Hence with in the same class we can declare 2 main methods with different arguments

Ex !-

```
Class Test
{
 p. s. v. m(String[] args)
 {
   S.o.pln(" String[] ");
 }
 public static void main(int[] args)
 {
   S.o.pln(" int[] ");
 }
}
```

o/p !- String[]

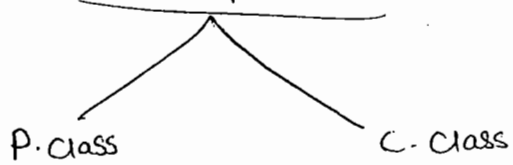
→ But JVM also

→ But Jvm always Call String arrguements main method only. The other main method we have to call explicitly just like a normal method call.

→ Inheritance Concept is applicable for static methods including main() method hence while executing child class if the child doesnot contain main method then the parent class main method will be execute

ex:-
```
Class p
{
  p.s.v.m(String[] args)
  {
    S.o.pln("parent Class");
  }
}
Class c extends p
{
}
```

Javac p.java



o/p java p
Parent class

o/p java c
parent class.

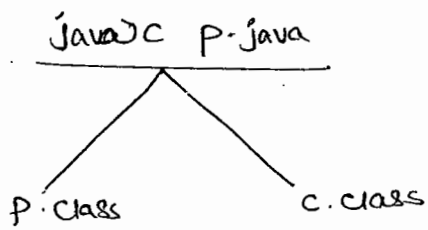
→ It Seems that overriding Concept is applicable for static methods but it is not overriding, it is method hiding.

ex:-
```
Class p
{
  p.s.v.m(——>
```

Eg:-

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

it is not overriding
it is method hiding



java P ←

parent main

java C ←

child main.

native modifier :-

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

→ The native methods are implemented in some other languages like C & C++ hence native methods also known as "foreign methods".

→ The main objectives of native keyword are ① to improve performance of the System

① To improve performance of the System.

② To use already existing legacy non-Java code.

Pseudo Code :-

→ To use native keyword

ex:-

```
class Native
{
  static
  {
    System.loadLibrary("native library")
  }
  public native void m1();
}

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

① Load native library

② Declare a native method

③ Invoke a native method

→ for native methods implementation is already available in other languages and we are not responsible to provide implementation. Hence native method declaration should compulsory Ends with ";"

Ex:- ① class Test

```
{
    public native void m1()
    {
    }
}
```

X

C.E :- native methods can't have a body.

② public native void m1();

① → for native methods implementation should be available in some other languages where as for abstract methods implementation should not be available hence abstract-native Combination is illegal Combination for methods.

② → native methods Cannot be declared with strictfp modifier because there no guarante that old language follows IEEE 754 Standard.

③ → Hence abstract native – strictfp Combination is illegal for methods.

→ the main disAdvantage of native keyword is it breaks platform independent nature of Java. because we are depending on result of platform dependent languages.

"Synchronized" modifier :-

→ Synchronized is the modifier applicable for methods & Blocks. we can't declare class & variable with this keyword.

→ If a method (or) Block declared as Synchronized then at a time only one Thread is allowed to operate on the given object.

→ The main advantage of Synchronized keyword is We can resolve data inconsistancy problems. But the main dis-Advantage of Synchronized keyword is it increases waiting time of thread and effects performance of the System. Hence, If there is no specific requirement it is never recommended to use Synchronized keyword.

"transient" modifier :-

→ transient is the modifier applicable only for variables & we can't apply for methods & classes.

→ At the time of Serialization, if we don't want to Save the value of a particular variable to meet Security Constraints, then we should go for transient keyword.

→ At the time of Serialization JVM ignores the original value of transient variable & default value will be Serialization.

"Volatile" modifier :-

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

→ If the value of a variable keep on changing Such type of variables we have to declare with volatile modifier.

↳ If a variable declared as volatile then for every thread a seperate local copy will be created.

↳ Every intermediate modification performed by that thread will takes place in local copy instead of master copy.

↳ Once the value got finalized just before terminating the thread the master copy value will be updated with local stable value.

↳ The main advantage of volatile keyword is we can resolve data inconsistency problems.

↳ But the main disAdvantage of volatile keyword is, creating & maintaining a seperate copy for every thread, increases complexity of the programing & effects performance of the System. Hence, if there is no specific requirement it is never recommended to use volatile keyword, & it is almost outdated keyword

↳ Volatile variable means it's value keep on changes where as 'final' variable means its value never changes. Hence final - volatile combination is illegal combination for variables.

Conclusion:

→ The only applicable modifier for local variables is final

→ The modifiers which are applicable only for variables but not for classes & methods are. Volatile & transient

→ The modifiers which are applicable only for methods but not for classes & variables native & synchronized.

→ The modifiers which are applicable for top level classes, methods & variables are public, <default>, final

Modifier	Classes		methods	variables	blocks	interfaces	enum	Constructors
	Outer	Inner						
public	✓	✓	✓	✓	×	✓	✓	✓
<default>	✓	✓	✓	✓	×	✓	✓	✓
private	×	✓	✓	✓	×	×	×	✓
protected	×	✓	✓	✓	×	×	×	✓
final	✓	✓	✓	✓	×	×	×	×
abstract	✓	✓	✓	×	×	✓	×	×
static	×	✓	✓	✓	✓	×	×	×
synchronized	×	×	✓	×	✓	×	×	×
native	×	×	✓	×	×	×	×	×
strictfp	✓	✓	✓	×	×	✓	✓	×
transient	×	×	×	✓	×	×	×	×
volatile	×	×	×	✓	×	×	×	×

→ The modifiers which are applicable for Inner classes but not for Outer classes are private, protected, Static

Interfaces

(1) Introduction

(2) Interface declaration & Implementation
   (a) extends Vs implements.

(3) Interface methods

(4) Interface Variables

(5) Interface Naming Conflicts
   (1) method Naming Conflicts
   (2) Variable " "

(6) Marker Interface

(7) Adapter class

(8) Abstract class Vs Concrete class vs interface.

(9) diff.. blw abstract class & interface

Interface :-

① Any Service Requirement Specification (SRS) is Considered as Interface.

→ from the client point of view an Interface defines the Set of Services what is Expecting.

→ from the Service provider point of view an Interface defines the Set of Services what is offering.

② Hence an Interface Considered as Contract b/w Client & Service provider

Ex:-

→ By using Bank ATM GUI Screen, Bank people will highLate the Set of Services what they are offering At the Same time the Same Screen describes the Set of Services what End-User is Expected.

Hence this GUI Screen acts as Contract b/w bank people & Customers

→ with in the Interface we Can't write any implementation; because it has to highLight Just the Set of Services what we are offering or what you are Expecting. Hence every method present inside interface Should be abstract. Due to this interface is Considered as 100% pure abstract class

What is an Interface :-

→ Any Service Requirement Specification (SRS) (or) Any Contract b/w Client & Service provider (or) 100% pure abstract class is nothing but an Interface.

→ The main Advantages of Interfaces are:

(i) We can acheive Security. because we are not highlighting our internal implementation.

(ii) Enhancement will become very easy, because without effecting outside person we can change our internal implementation.

(iii) Two different Systems can communicate via Interface (A Java application can talk with Mainframe System through Interface).

Declaration & Implementation of an Interface :-

→ We can declare an Interface by using Interface keyword, we can implement an Interface by using implements keyword.

Ex:-

```
interface Interf
{
  void m1();  // by default public abstract void m1();
  void m2();
}
abstract class ServiceProvider implements Interf
{
→ public void m1()
  {
  }
}
```

→ If a class implements an interface compulsary we should provide implementation for every method of that interface otherwise we have to declare class as abstract. Violation leads to Compile-time Error.

→ when ever we are implementing an interface method Compulsary it should be declared as public otherwise we will get CompileTimeError.

Extends Vs implements :-

1. A class can extend only one class at a time.
2. A class can implement any no. of interfaces at a time.
3. A class can extend a class and can implement any no. of interfaces simultaneously.
4. An Interface can extend any no. of interfaces at a time.

ex:-

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

Q) which of the following is True?

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

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

(3) A class can extend a class or can implement an interface but not both simultaneously ✗

(4) An Interface can extend only one interface at a time ✗

(5) An Interface can implement any no. of classes at a time ✗

(6) None of the above ✓

(Q) Consider the expression

X extends Y

— for which of the following possibilities this expression is True?

① Both should be classes

② Both should be interfaces

✓ ③ Both can be either classes or interfaces

④ No Restriction.

(A):

① X extends Y, Z

(a) X, Y, Z should be interfaces

② X extends Y implements Z

X, Y → classes

Z → interfaces

③ X implements Y extends Z ✗

→ C.E

Interface methods :-

Whether we are declaring or not, every interface method is by default, public & abstract

ex:-
```
interface Interf
{
    void m1();
}
```
public :-

→ To make this method availability for every implementation class.

abstract :-

Because interface methods specifies requirements but not implementation.

Hence the following method declarations are equal inside interface.

(1) void m1(); ✓

(2) public void m1(); ✓

(3) abstract void m1(); ✓

(4) public abstract void m1(); ✓

→ As every interface method is by default public & abstract the following modifiers are not applicable for interface methods.

(1) private
(2) protected
(3) <default>
(4) final

} ✗

(5) static
(6) strictfp
(7) synchronized
(8) native

} ✗

→ Which of the following method declaration are valied inside interface:

(1) public void m1() {} ✗

(2) public static void m1(); ✗

(3) public synchronized void m1(); ✗

(4) private abstract void m1(); ✗

(5) public abstract void m1(); ✓

Interface variables:-

→ An interface can contain variables the main purpose of these variables is to specify.

Constents at Requirement Level:-

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

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

public:- To make this variable available for every implementation class.

static:- without existing object also implementation class can access this variable.

final:- implementation class can access this variable but can't modify.

→ Hence inside interface the following declaration are valid & equal.

1) int x=10;
2) public int x=10;
3) public static int x=10;
4) public static final int x=10;
5) public static int x=10;
6) final int x=10;
7) public final int x=10;

8) Static final int x=10;

→ As interface variables are public static & final we can't declare with the following modifiers.

(1) private (3) <default> (5) volatile.
(2) protected (4) transient

→ for the interface variable compulsary are should perform initialization at the time of declaration only otherwise will get compile time error.

(*)
```
interface Interf
{
    int x; X    C.E:- = Expected.
}
```

→ which of the following variable declarations are allowed inside interface.

(1) int x=10; ✓
(2) int x; X
(3) private int x=10; X
(4) public int x=10; ✓
(5) transient int x=10; X
(6) volatile int x=10; X
(7) public static final int x=10; ✓

→ Inside implementation classes we can access interface variables but we can't modify these values.

Ex:-

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

```
Class Test implements Intef
{
 p.s.v.m(String[] args)
 {
    x=888;
    S.o.pln(x);
 }
}
```

C.E:

```
Class Test implements Intef
{
 p.s.v.m(String[] args)
 {
    int x=88;
    S.o.pln(x);88
 }
}
```

Interface Naming Conflicts:-

① Method naming Conflicts:-

Case1:-

→ If Two interfaces Contains a method with Same Signature & Same return type in the implementation class we can provide implementation for only one method.

Ex:-

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

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

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

Case 2 :-

→ If Two interfaces Contains a method with same name but different args then, in the implementation class we have to provide implementation for both methods & these methods are Considered as overloaded methods.

Eg:-

```
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

Case3 :-

→ If two interfaces Contains a method with Same Signature but different returntypes. Then it is impossible to implement both interfaces at a time.

Ex :-

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

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

→ We Cant write any Java Class which implements both interfaces Simotaneously.

Q) is It possible A Java class Can implement any no. of interfaces Simultaneously.

*) yes, Except if two interfaces Contains a method with Same Signature but different return types.

(2) Variable naming Conflicts :-

```
interface Left
{
  int x=888;
}
interface Right
{
  int x=999;
}
```

```
Class Test implements Left, Right
{
 p.s.v.m(——>)
 {
  S.o.pln(x);
 }
}
```

C.E:- reference to x is ambiguous.

→ There may be a chance of 2 interfaces contains variable with same name & may arise variable naming conflicts But we can resolve these naming conflicts by using interface names.

```
S.o.p(Left.x) ; 888
S.o.p(Right.x) ; 999
```

Marker Interface :-

Ex :- Kenya

→ If an interface wont contain any method & by implementing that interface if other objects will get ability such type of interfaces are called marker interface (or) Tag interface (or) ability interface.

Ex:- Serializable, Clonable, RandomAccess, SingleThreadModel.

⇒ these interfaces are marked from some ability.

Ex:- By implementing Serializable interface we can send objects across the N/w and we can save state of object to a file. This extra ability is provided through Seriable interface.

Ex:- By implementing Cloneable interface our object will be in a position to provide exactly duplicate objects

Q) Marker interface won't Contain any method then how the objects will get that Special ability?

A) JVM is responsible to provide required ability in marker interfaces.

Q) Why JVM is providing required ability in marker interface?

A) To reduce Complexity of the programming.

Q) Is it possible to Create our own Marker Interface?

A) Yes, But Customization of JVM is required.

Ex:- Sleepable, Eatable, Jumpable, Lovable, Funnable.

Adapter Class:-

→ Adapter class is a Simple java class that implements an interface, an interface only with Empty implementation.

```
interface X
{
  m1();
  m2();
  .
  .
  m1000();
}
```

```
abstract class Adapter X implements X
{
  m1(){}
  m2(){}
  .
  .
  m1000(){}
}
```

If we create an object for this Empty result so for this class we declare as abstract. by default abstract

→ If we implement an interface directly ~~con~~ Compulsary we should provide implementation for every method of that interface, whether we are interested or not & whether it is required (or) not. It increases length of the code, so that readability will be reduced.

```
Class Test implements X
{
  m1() {}
  m2() {}
  m3()
  {
  ==
  }
  !
  m100() {}
}
```

If we extends adapter class instead of implementation interface directly then we have to provide implementation of only for required method but not all this approach reduce length of the code & improves readability.

⇒ Class Test extends Adapter x

```
{
  m4()
  {
  ==
  }
}
```

Concreate class Vs abstract class Vs interface :-

→ we don't know any thing about implementation Just we have requirements Specification, then we should go for interface

Ex1. Seavlet.

→ we are talking about implementation but not Completly (Just partially implementation) then we should go for abstract Class.

Ex:- Generic – Servlet

HTTP_Servlet

→ we are talking about implementation Completly & ready to provide Service, Then we should go for Concrete Class.

Ex:- Our own Servlet.

Difference b/w interfaces & abstract class:-

interface	abstract class
1) If we don't know any thing about implementation Just we have requirement Specification. then we should go for interface.	1) If we are talking about implementation but not Completly (partially implementation) then we should go for abstract class.
2) Every method present inside interface is by default public & abstract.	2) every method present inside abstract class need not be public & abstract. we can take Concrete methods also.
3) The following modifiers are not allowed for interface methods: strictfp, protected, static, native private, final, synchronized,	3) there are no restrictions for Abstract class method modifiers i.e, we can use any modifier.

4) every variable present inside interface is public, static final, by default whether we are declare (or) not	4) abstract class variables need not be public, final static.
5) for the interface variables we can't declare the following modifiers private, protected, transient, volatile	5) there are no restriction for abstract class variable modifiers.
6) for the interface variables compulsory we should perform initialization at the time of declaration only	6) for the abstract class variables there is no restriction like performing initialization at the time of declaration
7) Inside interface we can't take instance & static blocks.	7) Inside abstract class we can take static block & instance blocks.
8) Inside Interface we can't take Constructor.	8) Inside abstract class we can take Constructor.

Q) Inside abstract class we can take Constructor but we can't create an object of abstract class, what is the need?

A) → abstract class Constructor will be executed whenever we are create child class object to perform initialization of parent class instance variable at parent level only and this Constructor meant for child object creation only

Q) Inside Interface every method should be abstract where as in abstract class also we can take only abstract methods Then what is the need of interface.?

A) → Interface purpose we can replace abstract class but it is not a good programming practice we are miss using the role of abstract class.

→ we should bring abstract class into the picture whenever we are talking about implementation.