

## 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 at most 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 Compiletime Error.

→ 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: Sou.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 ✓

### Case 2:->

If class B declared as public & the program name is A.java, then we will get Compiletime Error saying,

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

### Case 3:-

If we declare both A & B classes as public & name of the program is B.java then we will get Compiletime Error 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");

}

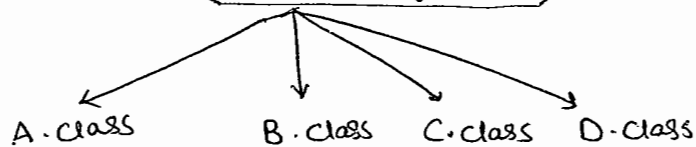
Class D

{

}

Save ⇒ Durga.java

javac 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:- NoSuchMethodException: main

⑤ Java Durga ←

R.E:- NOClassDefFoundException: Durga

Note 1.

→ 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:- Cannot find Symbol  
Symbol: class ArrayList  
Location: class Test

C-E:- symbol: method ArrayList

→ 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 qualified name hence it reduces improves readability & reduces Length of the Code.

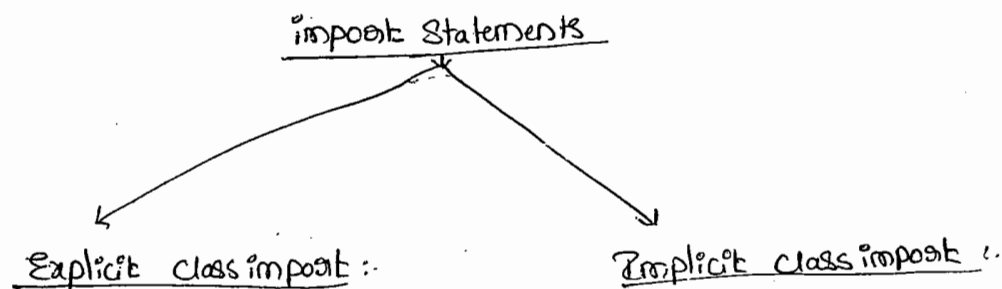
Case (1):-

Types of import Statements:-

→ There are 2 types of import Statements

(1) Explicit class import

(2) Implicit class import



Ex!- `import java.util.ArrayList;`

Ex!- `import java.util.*;`

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

→ Best Suitable for Hitech City where readability is important

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

→ Best Suitable for Ammerpet where typing is important.

### Case 2: Difference b/w #include & import Statement :-

- In C language #include all the specified header files will be loaded at the time of include statement only irrespective of whether we are using those header files or not. Hence this is "Static loading".
- But in the case of Java language import statement no <sup>class-</sup>file will be loaded at the time of import statement, in the next lines of code whenever we are loading a class at that time only the corresponding <sup>class-</sup>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?

- X ① import java.util;
- X ② import java.util.\*;
- ✓ ③ import java.util.\*;
- ✓ ④ import java.util.\*;

### Case 4:

→ Consider the Code,

```
class MyRemoteObject extends java.rmi.Unicast
                             RemoteObject
{
    ...
}
```

→ The code compiles fine even though 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 <http://javabynataraj.blogspot.com> it is not required.

Required — to use fully Qualified name.

Case:-

Sample:-

```
import java.util.*;
import java.sql.*;

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

Date  
List } available in both {util  
sql

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:-

```
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.

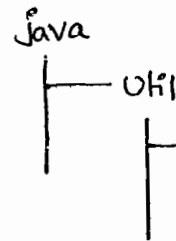
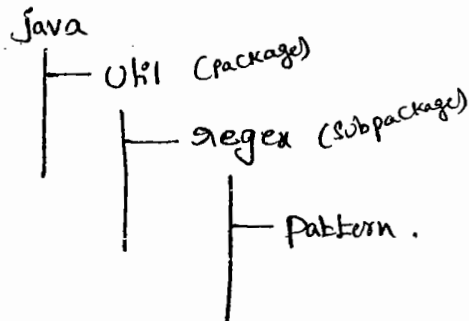
Conclusion:- While Resolving class names Compiler will always gives the precedence in the following order,

→ Order see above

### Case 7!.

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

Ex:-



→ To use PatternClass 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.



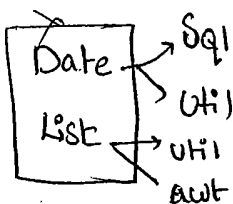
## Static import:-

- 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 whenever 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(Maths.sqrt(4));
        S.o.pln(Maths.random());
        S.o.pln(Maths.max(10, 20));
    }
}
```

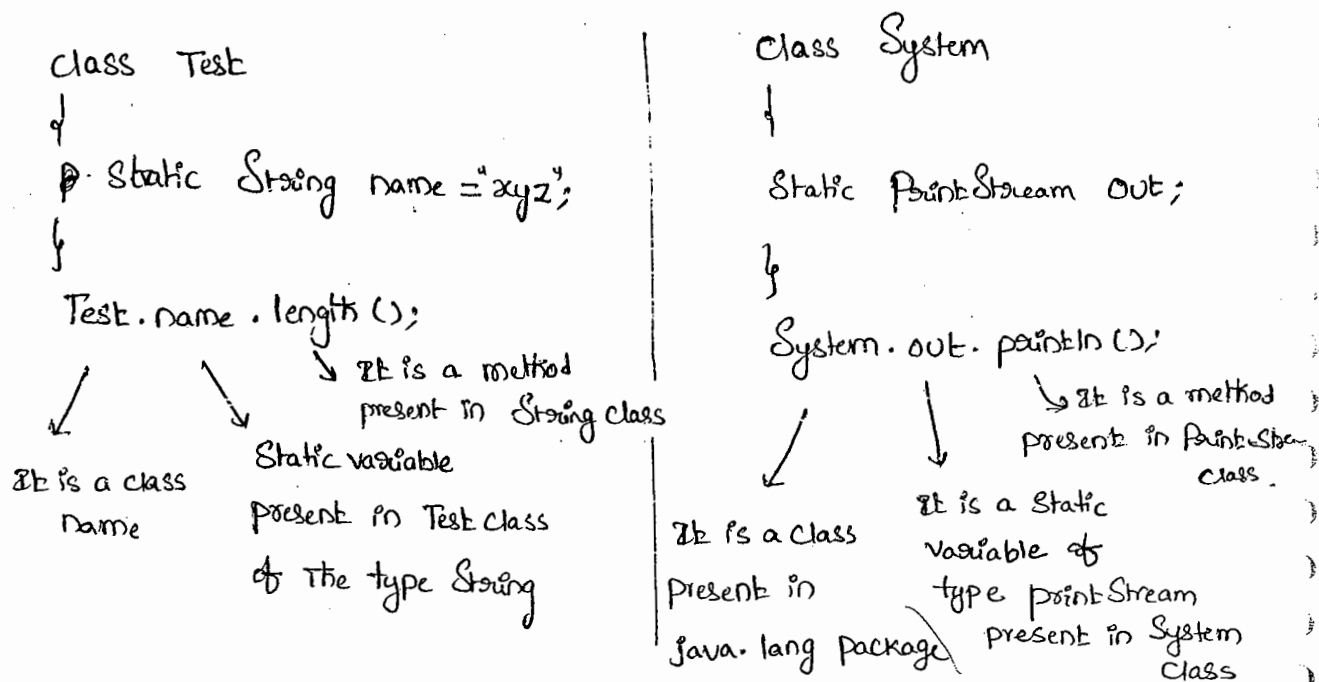


### With Static import

```
import static java.lang.math.sqrt;
import static java.lang.math.*;

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()` :-



Explanation:

→ `out` is a Static variable present in `System` class hence we can access by using classname.

→ But when even 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 perspective (Ambiguity)!

```
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 ambiguous

### 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 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.

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.

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

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

X ③ import static java.lang.math;

✓ ④ import java.lang.math;

✓ ⑤ import static java.lang.math.\*;

X ⑥ import static java.lang.math.Sqrt();

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

Normal import Vs Static import:-

→ we can use normal import to import classes & interfaces of a package. when ever 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 members we can access directly.

# Packages

9846 classes are  
there in java  
according to 1.6v

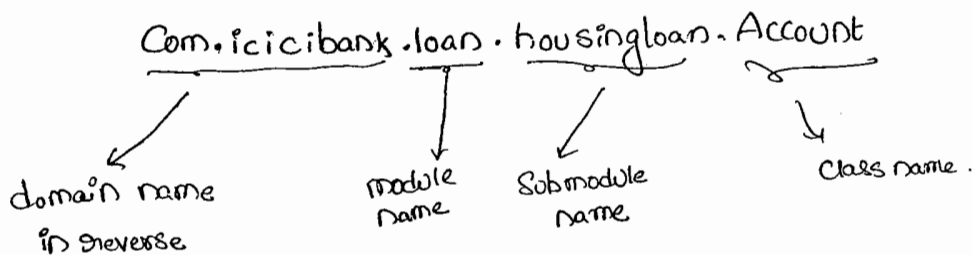
## Package:-

→ 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 convention to name packages i.e. to use internet domain name in reverse.



## Ex:-

```
Package Com.durgajobs.itjobs;
```

```
public class HdJobs
```

```
{
```

```
    P.S.V.m(String[] args)
```

```
{
```

```
    S.o.pln("Getting jobs is very easy");
```

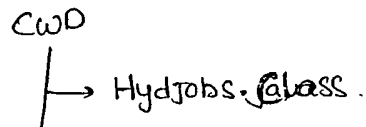
```
}
```

```
}
```

① `javac HydJobs.java`

Fl

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

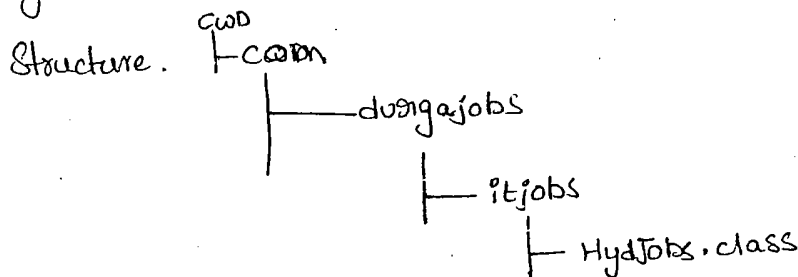


② `javac -d . HydJobs.java`

→ destination  
to place generated  
class files

current  
working  
directory

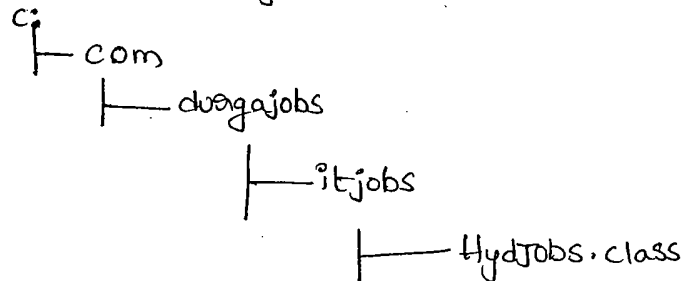
→ generated class file will be placed into corresponding package structure.



→ 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 <sup>already</sup> available then we will get Compile time Error

Ex: `javac -d z: HydJobs.java`

<http://javabynataraj.blogspot.com> 138 of 255.

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

Run

→ Java Com.durgaJobs. itjobs. HydJobs ←

o/p! Getting Job is Very easy.

### Conclusions:-

① → In Any Java program there should be only At most 1 package Statement. If we are taking more than 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).

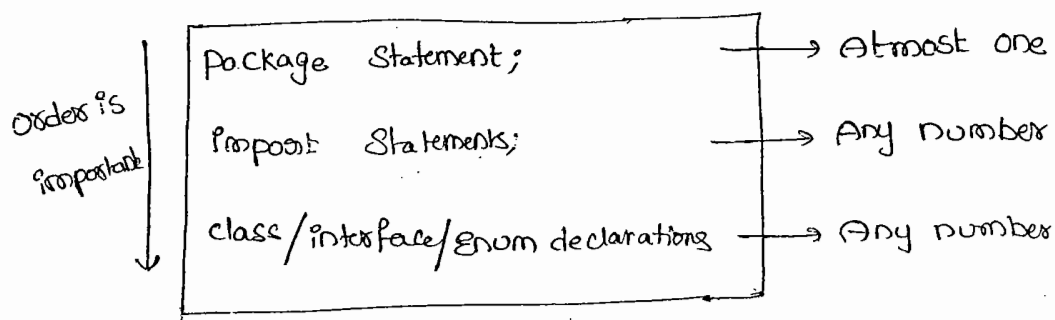
Ex:- ✓ import java.util.\*;

→ package pack1;  
    Class A  
    {  
    }  
    }

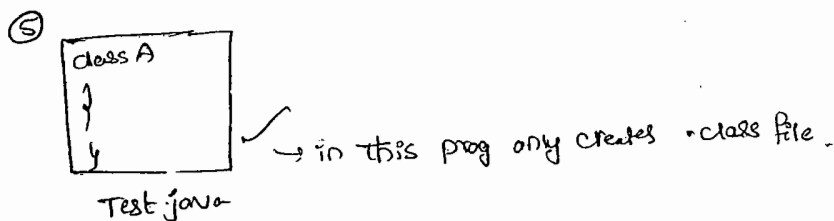
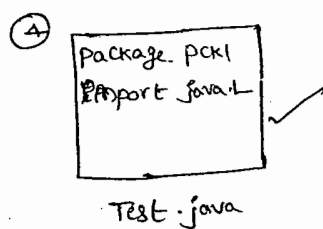
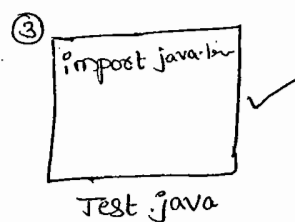
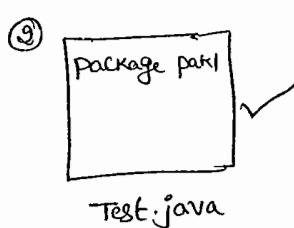
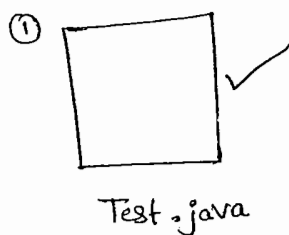
C.E:- class, interface or enum Expected.



→ The proper Structure of a Java Source file is



→ The following are Valid Java programs.



→ An Empty Source file is a Valid Java program.

## Class modifiers

→ when ever we are creating our own java class compulsory we have to provide some information about our class to the JVM like,

- ① wheather our class <sup>can be</sup> accessible from any where or not
- ② wheather child class creation is possible for our class or not.
- ③ wheather instantiation 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 Compiletime Error.  
Saying "modifier xxxxxx not allowed here".

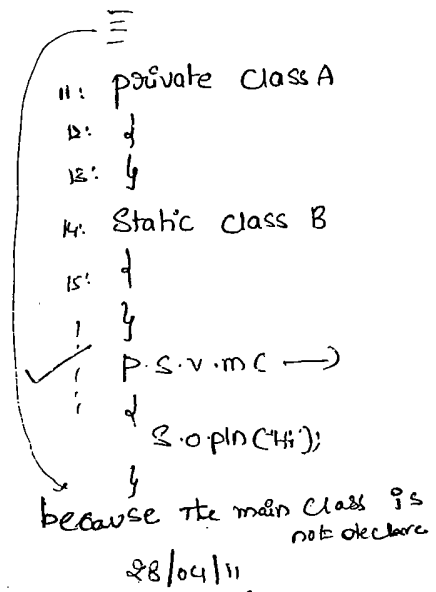
Ex:-  

```
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.



### 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:-

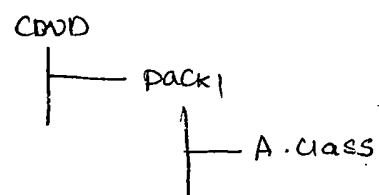
```

package pack1;

public class A
{
    public void m1()
    {
        S.O.P.L.N('Hello');
    }
}

```

Javac -d . A.java



```

package pack2;
import pack1.A;

class B
{
    P.S.V.m (String[] args)
    {
        A a = new A();
        a.m1();
    }
}

```

Comp! 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; Cant be accessed from outside package.

default classes :-

→ If a class declared as default then we can access that class only within 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 ~~to~~ override that method in the child class.

Ex:-

```

Class P
{
    public void property()
    {
        S.o.pln("money + Gold + Land");
    }
    public final void maanyu()
    {
        S.o.pln("Subba laxmi");
    }
}

C.E
{
    Class C extends P
    {
        public void maanyu()
        {
            S.o.pln("Kajal | Zsba | atara");
        }
    }
}
  
```

C.E!- maanyu() in C cannot override maanyu() in P; overrider 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
        {
        }
        {
        }
  
```

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 by default. but Every variable present in final class need not be final.
- The main Advantage of final keyword is we can achieve 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:-

- Even though we don't <sup>know about</sup> 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 compulsory 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 Vehicle

{

public abstract int getNoOfWheels();

}

Class Bus extends Vehicle

{

public int getNoOfWheels()

{

return 6;

}

}

Class Auto extends Vehicle

{

public int getNoOfWheels()

{

return 3;

}

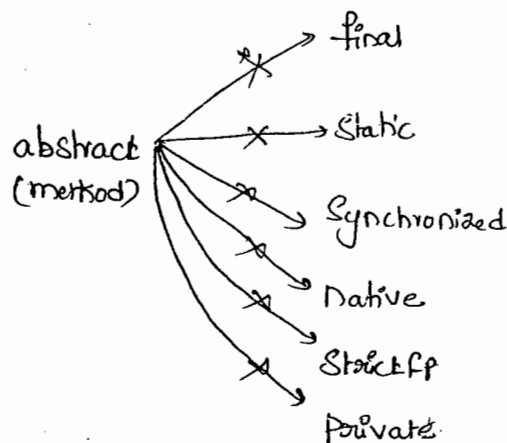
}

→ 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 instantiation then we have to declare that class as abstract. i.e., for abstract classes instantiation (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();

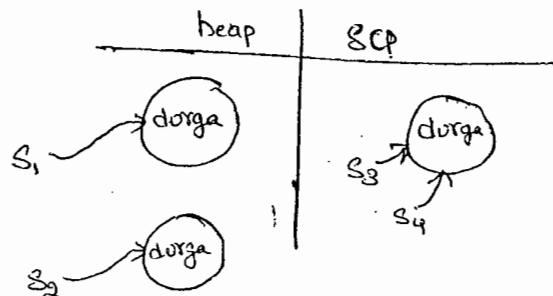


Ex:- String S<sub>1</sub> = new String("durga");

String S<sub>2</sub> = new String("durga");

String S<sub>3</sub> = "durga";

String S<sub>4</sub> = "durga";



ohm Sai Ram prabu.

durga sir.

- 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

- ✓ 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, Frame, Canvas)
- 13) Generics
- 14) Regular Expressions
- ✓ 15) G.C
- ✓ 16) Assertions (14)
- 17) J8N
- 18) enum
- 19) development

dell

chaithanyaobis@gmail.com

Satish4dwarid@

29444524

Chait

Chaithanya - Anumanchi@dell.com.

ON durgajobsInfo 9870867070

## abstract class Vs abstract method :-

77

→ If a class Contains atleast One abstract method then Compulsary That class should be declared as abstract otherwise we will get Compiletime Error. because, the implementation is not complete & hence we can't create an object.

→ Eventhough a 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:- HttpServlet, 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 have an 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:- SubTest is not abstract and does not override abstract method m2() in Test

→ we can handle these compiletime error either by declaring <sup>class</sup> SubTest as abstract or by providing implementation for m2().

Note:-

→ The usage of abstract methods, abstract class & interfaces are recommended & it is always good programming 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();
}

```

X

```

abstract class A
{
    public final void m1();
}

```

✓

Strictfp (all lower case) modifier :- (strictfloatingpoint)

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

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

→ strictfp <sup>method</sup> 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 concrete 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 Structfp void m1(); X (invalid)

### Member (variables & methods) modifiers :-

#### ① public members :-

→ If we declare a member as public then we can access that member from anywhere but corresponding 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();  
    }  
}
```

→ Even though m1() method is public, we can't access m1() from outside of pack1 because the corresponding 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 within 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 within the current class.
- abstract methods should be visible in child classes to provide implementation. whereas 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 within the current package anywhere but outside package only in child classes.

Protected  $\equiv$  <default> + kids of an another package (only child reference).

- \* within 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.

```

Ex1. 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.m();

✓ B b = new B()

✓ | b.m();

✓ A a1 = new B()

✓ | a1.m();

→ 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

Package pack2;

import pack1.A;

public class C extends A

{

p.s.v.m(—)

{

A a = new A()

X a.m();

C c = new C()

✓ c.m();

A a1 = new C()

X a1.m();

}

pack1  
A {  
 protected void m();

package2  
B extends A  
 | ✓  
C extends B  
 | ✓

package3  
D extends B  
 | ✓

→ The most restricted



\* private < default < protected < public

80

visibility	private	<default>	protected	public
① within the Same class	✓	✓	✓	✓
② from child class of Same Package	X	✓	✓	✓
③ from non-child class of Same package	X	✓	✓	✓
④ from child class of outside Package.	X	X	✓ <small>But we should use only child class reference</small>	✓
⑤ from non-child class of outside package.	X	X	X	✓

⇒ "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 provide any default values. Compulsary 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 compulsary we should perform initialization whether 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 <sup>have</sup> not been initialized.

Rule:-

① for the final instance variables we should perform initialization before Constructor Completion.

→ ie, 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;

}

C.E! Variable x might not have been initialized.

### Rule:-

\* For the final static variables we should perform initialization Before class loading completion.

\* i.e., the following are various places to perform this

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;  
    }  
}
```

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

class Test

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

C.E:- Can't assign a <sup>value</sup> variable to final variable x.

### iii) Final Local Variables :-

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

Ex:-  
 ① Class Test  
 {  
     public void main()  
     {  
         int x;  
         S.o.pln("Hello");  
     }  
 }  
 %p!- Hello

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

③ → Even though Local variable declared as the final it is not required to perform initialization if we are not using that variable.

Ex:-  
 Class Test  
 {  
     p.s.v.m()  
     {  
         final int x;  
         S.o.pln("Hello Sai");  
     }  
 }  
 %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; X  
 protected int x=30; X  
 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 Compile-time Error.

Ex:-

Class Test

{

p.s v.m( )

{

m(10, 20);

}

Actual parameters

p.s.v.m( final int x, int y )

{

formal parameters

x=1000;

y=2000;

// Can't assign a value to final variable x.

s.o.println( x + " + " + y );

}

Static → class level  
instance → object level

### Static modifier:-

→ Static is the modifier applicable for variables & methods but not for classes (but inner class 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 separate 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.

first static variable is created at the beginning when class is created.

19

Ex:-

Class Test

{

int x=10;

Static int y=20;

P.S.V.M(—)

{

Test t<sub>1</sub> = new Test();t<sub>1</sub>.x = 888;t<sub>1</sub>.y = 999;Test t<sub>2</sub> = new Test();S.o.pln(t<sub>1</sub>.x + "----" + t<sub>2</sub>.y);

{

10

999

{

1st  
y=202nd  
x=10

y=20

y=20  
999

x=10

for every object a  
separate copy will be  
created.

→ 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 & II

✗ B). I & IV. LE!:- non-static variable  $\alpha$  cannot be accessed from static context

✓ C). II & III

✓ D). II & IV

✗ E). I & II

✗ F). III & IV

→ 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.

→ 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

{

public static void m(String[] args)

{

System.out.println("String[]");

}

public static void main(int[] args)

{

System.out.println("int[]");

}

}

//p!:- String[]

→ But JVM also



→ But Jvm always Call Strongly argumented 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 doesn't contain main method then the parent class main method will be executed.

ex-  
 Class P  
 {  
 P.S.V.M (String[] args)  
 {  
 S.o.pln ("Parent Class");  
 }  
 }

Class C extends P

Javac P.java

P.class

C.class

%p Java P

%p Java C

Parent class

parent class.

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

ex-1-

class P

{

P.S.V.M ( )

Ex:.

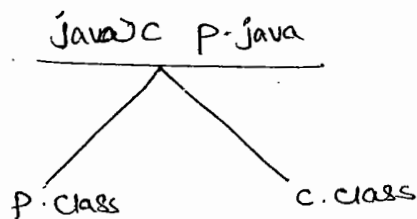
```

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 :-

85

- 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
```

① Load native library

```
    {  
        System
```

```
        System.loadLibrary("native Library")  
    }
```

② Declare

a native method

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

```
class Child
```

```
    {  
        p.s.v.m(—)
```

③ Invoke a Native n = new Native();

Native method n.m1();

```
    }
```

→ 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()

{

}

}

CE!:- Native methods Can't have a body.

② public native void m1(); ✓

① → For native methods implementation should be available in some other languages whereas 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 is no guarantee 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.

### ⑥ "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 inconsistency problems. But the main disadvantage of Synchronized keyword is it increases waiting time of thread and affects performance of the system.
- Hence, if there is no specific requirement it is never recommended to use Synchronized keyword.

### ⑦ "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.

### ⑧ "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.

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

→ Every intermediate modification performed by that thread will take 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 ~~avoid~~ resolve data inconsistency problems.

→ But the main disadvantage of volatile keyword is, creating & maintaining a separate copy for every thread, increases complexity of the program & affects 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 its value keep on changes whereas '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 are native & synchronized.

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

modifier	Classes		variables	blocks	interfaces	enum	Constructors
	Outer	Inner					
Public	✓	✓	✓	×	✓	✓	✓
<default>	✓	✓	✓	×	✓	✓	✓
Private	×	✓	✓	×	×	×	✓
Protected	×	✓	✓	×	×	×	✓
final	✓	✓	✓	×	×	×	×
abstract	✓	✓	×	×	✓	×	×
Static	✗	✓	✓	✓	×	×	×
Synchronized	×	×	×	✓	×	×	×
Native	×	×	×	×	×	×	×
Sticky	✓	✓	×	×	✓	✓	×
transient	×	×	✓	×	×	×	×
Volatile	×	×	✓	×	×	×	×