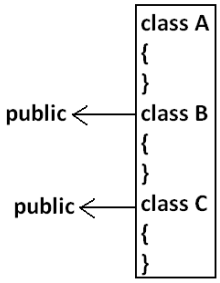
Declaration and Access Modifiers

1) Java source file structure
2) Class modifiers
3) Member modifiers
4) Interfaces

Java source file structure:

- A java program can contain any no. Of classes but at mot one class can be declared as public. "If there is a public class the name of the program and name of the public class must be matched otherwise we will get compile time error".
- If there is no public class then any name we gives for java source file.

Example:



Case1:

- If there is no public class then we can use any name for java source file there are no restrictions.

Example:

A.java
B.java
C.java
Bhaskar.java

case2:

- If class B declared as public then the name of the program should be B.java otherwise we will get compile time error saying "class B is public, should be declared in a file named B.java".

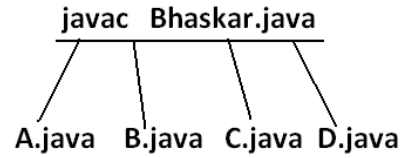
Cae3:

- If both B and C classes are declared as public and name of the file is B.java then we will get compile time error saying "class C is public, should be declared in a file named C.java".
- It is highly recommended to take only one class for source file and name of the program (file) must be same as class name. This approach improves readability and understandability of the code.

Example:

```
class A
{
public static void main(String args[]){
System.out.println("A class main method is executed");
}
}
class B
{
public static void main(String args[]){
System.out.println("B class main method is executed");
}
}
class C
{
public static void main(String args[]){
System.out.println("C class main method is executed");
}
}
class D
{
}
```

Output:




```
D:\Java>java A
A class main method is executed
D:\Java>java B
B class main method is executed
D:\Java>java C
C class main method is executed
D:\Java>java D
Exception in thread "main" java.lang.NoSuchMethodError: main
D:\Java>java Bhaskar
Exception in thread "main" java.lang.NoClassDefFoundError: Bhaskar
```

- We can compile a java program but not java class in that program for every class one dot class file will be created.

- We can run a java class but not java source file whenever we are trying to run a class the corresponding class main method will be executed.
- If the class won't contain main method then we will get runtime exception saying "NoSuchMethodError: main".
- If we are trying to execute a java class and if the corresponding .class file is not available then we will get runtime execution saying "NoClassDefFoundError: Bhaskar".

Import statement:

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

Output:

```
Compile time error.
D:\Java>javac Test.java
Test.java:3: cannot find symbol
symbol  : class ArrayList
location: class Test
ArrayList l=new ArrayList();
Test.java:3: cannot find symbol
symbol  : class ArrayList
location: class Test
ArrayList l=new ArrayList();
```

- We can resolve this problem by using fully qualified name "java.util.ArrayList l=new java.util.ArrayList();". But problem with using fully qualified name every time is it increases length of the code and reduces readability.
- We can resolve this problem by using import statements.

Example:

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

Output:

```
D:\Java>javac Test.java
```

- Hence whenever we are using import statement it is not require to use fully qualified names we can use short names directly. This approach decreases length of the code and improves readability.

Case 1: Types of Import Statements:

- There are 2 types of import statements.
    1) Explicit class import
    2) Implicit class import.

Explicit class import:

Example: Import java.util.ArrayList

- This type of import is highly recommended to use because it improves readability of the code.
- Best suitable for Hi-Tech city where readability is important.

Implicit class import:

Example: import java.util.*;

- It is never recommended to use because it reduces readability of the code.
- Bet suitable for Ameerpet where typing is important.

Case2:

Which of the following import statements are valid?

import java.util; ×
import java.util.ArrayList.*; ×
import java.util.*; ✓
import java.util.ArrayList; ✓

Case3:

- consider the following code.

```
class MyArrayList extends java.util.ArrayList
{
}
```

- The code compiles fine even though we are not using import statements because we used fully qualified name.
- Whenever we are using fully qualified name it is not required to use import statement. Similarly whenever we are using import statements it is not require to use fully qualified name.

Case4:

Example:

```
import java.util.*;
import java.sql.*;
class Test
{
```

```
public static void main(String args[])
{
Date d=new Date();
}}
```

Output:

```
Compile time error.
D:\Java>javac Test.java
Test.java:7: reference to Date is ambiguous, both class java.sql.Date in java.sql and class
java.util.Date in java.util match
Date d=new Date();
```

Note: Even in the List case also we may get the same ambiguity problem because it is available in both UTIL and AWT packages.

Case5:

- While resolving class names compiler will always gives the importance in the following order.
    1) Explicit class import
    2) Classes present in current working directory.
    3) Implicit class import.

Example:

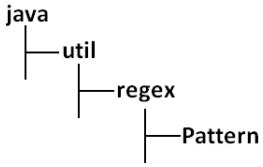
```
import java.util.Date;
import java.sql.*;
class Test
{
public static void main(String args[]){
Date d=new Date();
}}
```

- The code compiles fine and in this case util package Date will be considered.

Case6:

- Whenever we are importing a package all classes and interfaces present in that package are by default available but not sub package classes.

Example:



To use pattern class in our program directly which import statement is required?

1.import java.*; ✗
2.import java.util.*; ✗
3.import java.util.regex.*; ✓
4.import java.util.regex.Pattern; ✓

Case7:

- In any java program the following 2 packages are not require to import because these are available by default to every java program.
    1. java.lang package
    2. default package(current working directory)

Case8:

- "Import statement is totally compile time concept" if more no of imports are there then more will be the compile time but there is "no change in execution time".

Difference between C language #include and java language import.

- In the case of C language #include all the header files will be loaded at the time of include statement hence it follows static loading.
- But in java import statement no ".class" will be loaded at the time of import statements in the next lines of the code whenever we are using a particular class then only corresponding ".class" file will be loaded. Hence it follows "dynamic loading" or "load-on –demand" or "load-on-fly".

Static import:

- This concept introduced in 1.5 versions. According to sun static import improves readability of the code but according to worldwide programming exports (like us) static imports creates confusion and reduces readability of the code. Hence if there is no specific requirement never recommended to use a static import.

1.5 versions new features

1) For-Each
2) Var-arg
3) Queue
4) Generics
5) Auto boxing and Auto unboxing
6) Co-varient return types
7) Annotations
8) Enum
9) Static import
10) String builder

- Usually we can access static members by using class name but whenever we are using static import it is not require to use class name we can access directly.

Without static import:

class Test

```
{
public static void main(String args[]){
System.out.println(Math.sqrt(4));
System.out.println(Math.max(10,20));
System.out.println(Math.random());
}}
```

Output:

```
D:\Java>javac Test.java
D:\Java>java Test
2.0
20
0.841306154315576
```

With static import:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.*;
class Test
{
public static void main(String args[]){
System.out.println(sqrt(4));
System.out.println(max(10,20));
System.out.println(random());
}}
```

Output:

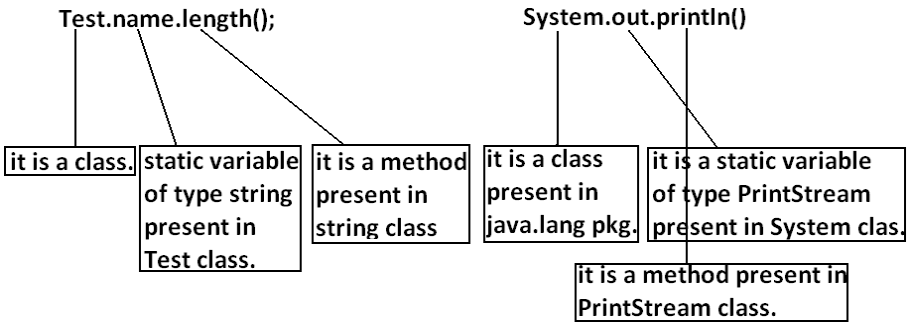
```
D:\Java>javac Test.java
D:\Java>java Test
2.0
20
0.4302853847363891
```

Explain about System.out.println statement?

Example 1 and example 2:

```
1)
class Test
{
static String name="bhaskar";
}
```

```
2)
import java.io.*;
class System
{
static PrintStream out;
}
```



Example 3:

```
import static java.lang.System.out;
class Test
{
public static void main(String args[]){
out.println("hello");
out.println("hi");
}}
```

Output:

```
D:\Java>javac Test.java
D:\Java>java Test
hello
hi
```

Example 4:

```
import static java.lang.Integer.*;
import static java.lang.Byte.*;
class Test
{
public static void main(String args[]){
System.out.println(MAX_VALUE);
}}
```

Output:

Compile time error.

```
D:\Java>javac Test.java
Test.java:6: reference to MAX_VALUE is ambiguous, both variable MAX_VALUE in java.lang.Integer and variable MAX_VALUE in java.lang.Byte match
System.out.println(MAX_VALUE);
```

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

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

- While resolving static members compiler will give the precedence in the following order.
    1. Current class static members
    2. Explicit static import
    3. implict static import.

Example:

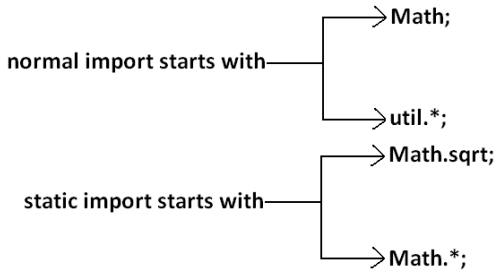
```
//import static java.lang.Integer.MAX_VALUE;------> line2
import static java.lang.Byte.*;
class Test
{
//static int MAX_VALUE=999;------> line1
public static void main(String args[])throws Exception{
System.out.println(MAX_VALUE);
}
}
```

- If we comet line one then we will get Integer class MAX_VALUE 2147483647.
- If we comet lines one and two then Byte class MAX_VALUE will be considered 127.

Which of the following import statements are valid?

1.import java.lang.Math.*; ×
2.import static java.lang.Math.*; ✓
3.import java.lang.Math; ✓
4.import static java.lang.Math; ×
5.import static java.lang.Math.sqrt.*; ×
6.import java.lang.Math.sqrt; ×
7.import static java.lang.Math.sqrt(); ×
8.import static java.lang.Math.sqrt; ✓

Diagram:



- Usage of static import reduces readability and creates confusion hence if there is no specific requirement never recommended to use static import.

What is the difference between general import and static import?

- We can use normal imports to import classes and interfaces of a package. whenever we are using normal import we can access class and interfaces directly by their short name it is not require to use fully qualified names.

- We can use static import to import static members of a particular class. whenever we are using static import it is not require to use class name we can access static members directly.

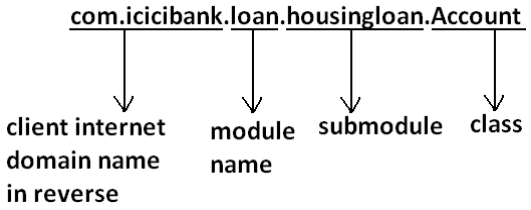
Package statement:

- It is an encapsulation mechanism to group related classes and interfaces into a single module.

The main objectives of packages are:

  - To resolve name confects.
  - To improve modularity of the application.
  - To provide security.
- There is one universally accepted naming conversion for packages that is to use internet domain name in reverse.

Example:



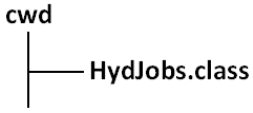
How to compile package program:

Example:

```
package com.durgajobs.itjobs;
class HydJobs
{
public static void main(String args[]){
System.out.println("package demo");
}
}
```

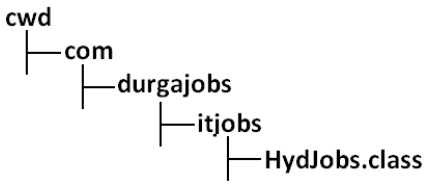
- Javac HydJobs.java generated class file will be placed in current working directory.

Diagram:



- Javac –d . HydJobs.java
- -d means destination to place generated class files "." means current working directory.
- Generated class file will be placed into corresponding package structure.

Diagram:



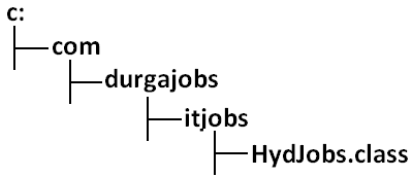
- If the specified package structure is not already available then this command itself will create the required package structure.
- As the destination we can use any valid directory.

If the specified destination is not available then we will get compile time error.

Example:

D:\Java>javac -d c: HydJobs.java

Diagram:



- If the specified destination is not available then we will get compile time error.

Example:

D:\Java>javac -d z: HydJobs.java

- If Z: is not available then we will get compile time error.

How to execute package program:

D:\Java>java com.durgajobs.itjobs.HydJobs

- At the time of execution compulsory we should provide fully qualified name.

Conclusion 1:

- In any java program there should be at most one package statement that is if we are taking more than one package statement we will get compile time error.

Example:

```
package pack1;
package pack2;
class A
{
}
```

Output:

Compile time error.

```
D:\Java>javac A.java
A.java:2: class, interface, or enum expected
package pack2;
```

Conclusion 2:

- In any java program the 1st non cement statement should be package statement [if it is available] otherwise we will get compile time error.

Example:

```
import java.util.*;
package pack1;
class A
{
}
```

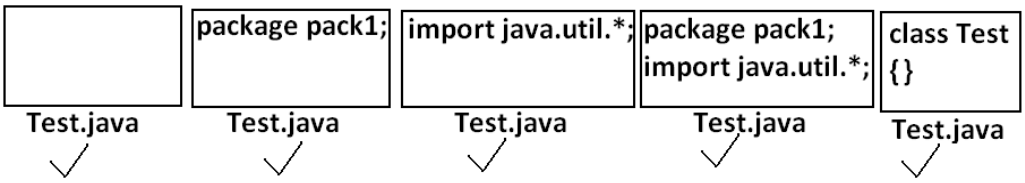
Output:

Compile time error.

```
D:\Java>javac A.java
A.java:2: class, interface, or enum expected
package pack1;
```

Java source file structure:

- All the following are valid java programs.



Note: An empty source file is a valid java program.

Class Modifiers

- Whenever we are writing our own classes compulsory we have to provide some information about our class to the jvm. Like
  1) Better this class can be accessible from anywhere or not.
  2) Better child class creation is possible or not.
  3) Whether object creation is possible or not etc.
- We can specify this information by using the corresponding modifiers.
- 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 compile time error.

Example:

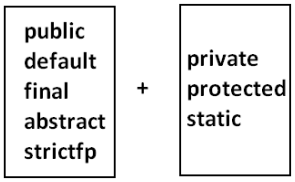
```
private class Test
{
public static void main(String args[]){
int i=0;
for(int j=0;j<3;j++)
{
i=i+j;
}
System.out.println(i);
}}
```

OUTPUT:

```
Compile time error.
D:\Java>javac Test.java
Test.java:1: modifier private not allowed here
private class Test
```

- But For the inner classes the following modifiers are allowed.

Diagram:



What is the difference between access specifier and access modifier?

- In old languages 'C' (or) 'C++' **public**, **private**, **protected**, **default** are considered as access specifiers and all the remaining 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 public then we can access that class from anywhere.

EXAMPLE:

Program1:

```
package pack1;
public class Test
{
public void methodOne(){
System.out.println("test class methodone is executed");
}}
```

Compile the above program:

```
D:\Java>javac -d . Test.java
```

Program2:

```
package pack2;
import pack1.Test;
class Test1
{
public static void main(String args[]){
Test t=new Test();
t.methodOne();
}}
```

OUTPUT:

```
D:\Java>javac -d . Test1.java
D:\Java>java pack2.Test1
Test class methodone is executed.
```

- If class Test is not public then while compiling Test1 class we will get compile time error saying **pack1.Test is not public in pack1; cannot be accessed from outside package**.

Default Classes:

- If a class declared as the **default** then we can access that class only **within the current package** hence default access is also known as "**package level access**".

Example:

Program 1:

```
package pack1;
class Test
{
public void methodOne(){
System.out.println("test class methodone is executed");
}}
```

Program 2:

```
package pack1;
import pack1.Test;
class Test1
{
public static void main(String args[]){
Test t=new Test();
t.methodOne();
}}
```

OUTPUT:

```
D:\Java>javac -d . Test.java
D:\Java>javac -d . Test1.java
```

D:\Java>java pack1.Test1
Test class methodone is executed

Final Modifier:

- Final is the modifier applicable for classes, methods and variables.

Final Methods:

- Whatever the methods parent has by default available to the child.
- If the child is not allowed to override any method, that method we have to declare with final in parent class. That is final methods cannot overridden.

Example:

Program 1:

```
class Parent
{
public void property(){
System.out.println("cash+gold+land");
}
public final void marriage(){
System.out.println("subbalakshmi");
}}
```

Program 2:

```
class child extends Parent
{
public void marriage(){
System.out.println("Thamanna");
}}
```

OUTPUT:

Compile time error.

D:\Java>javac Parent.java

D:\Java>javac child.java

child.java:3: marriage() in child cannot override marriage() in Parent; overridden method is final

public void marriage(){

Final Class:

- If a class declared as the final then we cann't creates the child class that is inheritance concept is not applicable for final classes.

EXAMPLE:

Program 1:

```
final class Parent
{
```

}

Program 2:

```
class child extends Parent
{
}
```

OUTPUT:

Compile time error.

```
D:\Java>javac Parent.java
D:\Java>javac child.java
```

child.java:1: cannot inherit from final Parent

```
class child extends Parent
```

- Note: Every method present inside a final class is always final by default whether we are declaring or not. But every variable present inside a final class need not be final.

Example:

```
final class parent
{
static int x=10;
static
{
x=999;
}}
```

- The main advantage of final keyword is we can achieve security. Whereas the main disadvantage is we are missing the key benefits of oops: polymorsim (because of final methods), inheritance (because of final classes) hence if there is no specific requirement never recommended to use final keyboard.

Abstract Modifier:

- Abstract is the modifier applicable only for methods and classes but not for variables.

Abstract Methods:

- Even though we don't have implementation still we can declare a method with abstract modifier. That is abstract methods have only declaration but not implementation. Hence abstract method declaration should compulsory ends with semicolon.

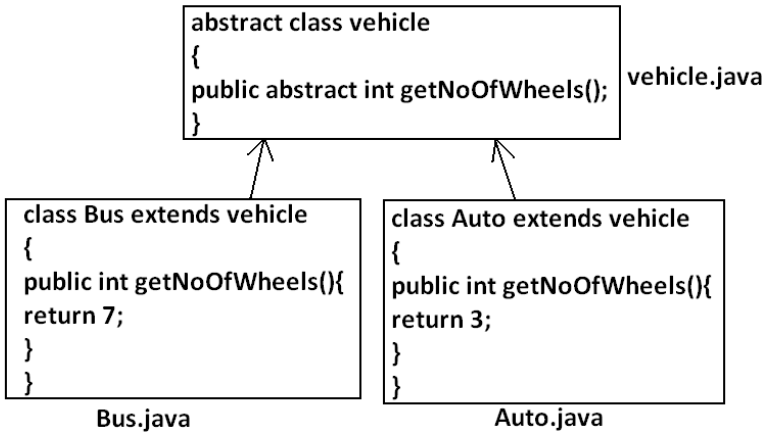
EXAMPLE:

public abstract void methodOne(); ⟶ valid
public abstract void methodOne(){} ⟶ invalid

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

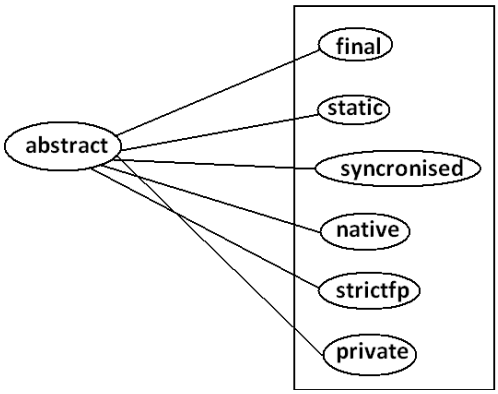
EXAMPLE:

PROGRAM:



- The main advantage of abstract methods is , by declaring abstract method in parent class we can provide guide lines to the child class such that which methods they should compulsory implement.
- Abstract method never talks about implementation whereas if any modifier talks about implementation it is always illegal combination.
- The following are the various illegal combinations for methods.

Diagram:



- All the 6 combinations are illegal.

Abstract class:

- For any java class if we are not allow to create an object such type of class we have to declare with abstract modifier that is for abstract class instantiation is not possible.

Example:

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

Output:

Compile time error.
D:\Java>javac Test.java
Test.java:4: Test is abstract; cannot be instantiated
Test t=new Test();

What is the difference between abstract class and abstract method?

- If a class contain at least on abstract method then compulsory the corresponding class should be declare with abstract modifier. Because implementation is not complete and hence we can't create object of that class.
- Even though class doesn't contain any abstract methods still we can declare the class as abstract that is an abstract class can contain zero no of abstract methods also.

Example1: HttpServlet class is abstract but it doesn't contain any abstract method.
Example2: Every adapter class is abstract but it doesn't contain any abstract method.

Example1:

```
class Parent
{
public void methodOne();
}
```

Output:

Compile time error.
D:\Java>javac Parent.java
Parent.java:3: missing method body, or declare abstract
public void methodOne();

Example2:

```
class Parent
{
public abstract void methodOne(){}
}
```

Output:

Compile time error.
Parent.java:3: abstract methods cannot have a body
public abstract void methodOne(){}

Example3:

```
class Parent
{
public abstract void methodOne();
}
```

Output:

Compile time error.

D:\Java>javac Parent.java
Parent.java:1: Parent is not abstract and does not override abstract method methodOne() in Parent
class Parent

- If a class extends any abstract class then compulsory we should provide implementation for every abstract method of the parent class otherwise we have to declare child class as abstract.

Example:

```
abstract class Parent
{
public abstract void methodOne();
public abstract void methodTwo();
}
class child extends Parent
{
public void methodOne(){}
}
```

Output:

Compile time error.
D:\Java>javac Parent.java
Parent.java:6: child is not abstract and does not override abstract method methodTwo() in Parent
class child extends Parent

- If we declare class child as abstract then the code compiles fine but child of child is responsible to provide implementation for methodTwo().

What is the difference between final and abstract?

- For abstract methods compulsory we should override in the child class to provide implementation. Whereas for final methods we can't override hence abstract final combination is illegal for methods.
- For abstract classes we should compulsory create child class to provide implementation whereas for final class we can't create child class. Hence final abstract combination is illegal for classes.
- Final class cannot contain abstract methods whereas abstract class can contain final method.

Example:

```
final class A
{
public abstract void
methodOne();
}
```
invalid

```
abstract class A
{
public final void methodOne(){
}
}
```
valid

Note:

- Usage of abstract methods, abstract classes and interfaces is always good programming practice.

Strictfp:

- strictfp is the modifier applicable for methods and classes but not for variables.
- Strictfp modifier introduced in 1.2 versions.
- If a method declare as the Strictfp then all the floating point calculations in that method has to follow IEEE754 standard. So that we will get flat from independent results.

Example:

```
System.out.println(10.0/3);
P4                  P3          IEEE754
3.33333333333333    3.333333    3.333
```

- If a class declares as the Strictfp then every concrete method(which has body) of that class has to follow IEEE754 standard for floating point arithmetic.

What is the difference between abstract and strictfp?

- Strictfp method talks about implementation where as abstract method never talks about implementation hence **abstract, strictfp** combination is illegal for methods.
- But we can declare a class with abstract and strictfp modifier simultaneously. That is abstract strictfp combination is legal for classes but illegal for methods.

Example:

```
public abstract strictfp void methodOne(); (invalid)
abstract strictfp class Test  (valid)
{
}
```

Member modifiers:

Public members:

- If a member declared as the public then we can access that member from anywhere "but the corresponding class must be visible" hence before checking member visibility we have to check class visibility.

Example:

Program 1:

```
package pack1;
```

```
class A
{
public void methodOne(){
System.out.println("a class method");
}}
```

D:\Java>javac -d . A.java

Program 2:

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

Output:

Compile time error.

D:\Java>javac -d . B.java

B.java:2: pack1.A is not public in pack1; cannot be accessed from outside package

import pack1.A;

- In the above program even though methodOne() method is public we can't access from class B because the corresponding class A is not public that is both classes and methods are public then only we can access.

Default member:

- If a member declared as the default then we can access that member only within the current package hence default member is also known as package level access.

Example 1:

Program 1:

```
package pack1;
class A
{
void methodOne(){
System.out.println("methodOne is executed");
}}
```

Program 2:

```
package pack1;
import pack1.A;
class B
```

```
{
public static void main(String args[]){
A a=new A();
a.methodOne();
}}
```

Output:

```
D:\Java>javac -d . A.java
D:\Java>javac -d . B.java
D:\Java>java pack1.B
methodOne is executed
```

Example 2:

Program 1:

```
package pack1;
class A
{
void methodOne(){
System.out.println("methodOne is executed");
}}
```

Program 2:

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

Output:

```
Compile time error.
D:\Java>javac -d . A.java
D:\Java>javac -d . B.java
B.java:2: pack1.A is not public in pack1; cannot be accessed from outside package
import pack1.A;
```

Private members:

- If a member declared as the private then we can access that member only with in the current class.

- Private methods are not visible in child classes where as abstract methods should be visible in child classes to provide implementation hence **private**, **abstract** combination is illegal for methods.

Protected members:

- If a member declared as the protected then we can access that member within the current package anywhere but outside package only in child classes.
- Protected=default+kids.
- We can access protected members within the current package anywhere either by child reference or by parent reference but from outside package we can access protected members only in child classes and should be by child reference only that is we can't use parent reference to call protected members from outside language.

Example:

Program 1:

```
package pack1;
public class A
{
protected void methodOne(){
System.out.println("methodOne is executed");
}}
```

Program 2:

```
package pack1;
class B extends A
{
public static void main(String args[]){
A a=new A();
a.methodOne();
B b=new B();
b.methodOne();
A a1=new B();
a1.methodOne();
}}
```

Output:

```
D:\Java>javac -d . A.java
D:\Java>javac -d . B.java
D:\Java>java pack1.B
methodOne is executed
methodOne is executed
methodOne is executed
```

Example 2:

```
package pack2;
import pack1.A;
public class C extends A
{
public static void main(String args[]){
A a=new A();
a.methodOne();
C c=new C();
c.methodOne();
A a1=new B();
a1.methodOne();
}
}
```

output:

```
compile time error.
D:\Java>javac -d . C.java
C.java:7: methodOne() has protected access in
pack1.A
a.methodOne();
```

Compression of private, default, protected and 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 child reference only	✓
5)From non-child class of outside package	✗	✗	✗	✓

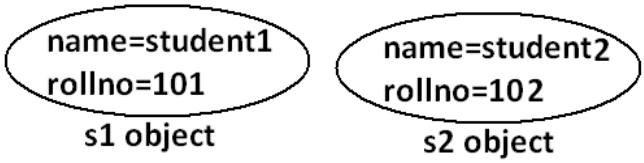
- The least accessible modifier is private.
- The most accessible modifier is public.
- Private<default<protected<public.
- Recommended modifier for variables is private where as recommended modifier for methods is public.

Final variables:

Final instance variables:

- If the value of a variable is varied from object to object such type of variables are called instance variables.
- For every object a separate copy of instance variables will be created.

DIAGRAM:



- For the instance variables it is not required to perform initialization explicitly jvm will always provide default values.

Example:

```
class Test
{
int i;
public static void main(String args[]){
Test t=new Test();
System.out.println(t.i);
}}
```

Output:

```
D:\Java>javac Test.java
D:\Java>java Test
0
```

- If the instance variable declared as the final compulsory we should perform initialization whether we are using or not otherwise we will get compile time error.

Example:

Program 1:

```
class Test
{
int i;
}
```

Output:

```
D:\Java>javac Test.java
D:\Java>
```

Program 2:

```
class Test
{
final int i;
}
```

Output:

```
Compile time error.
D:\Java>javac Test.java
```

Test.java:1: variable i might not have been initialized
class Test

Rule:

- For the final instance variables we should perform initialization before constructor completion. That is the following are various possible places for this.

1) At the time of declaration:

Example:

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

Output:

```
D:\Java>javac Test.java
D:\Java>
```

2) Inside instance block:

Example:

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

Output:

```
D:\Java>javac Test.java
D:\Java>
```

3) Inside constructor:

Example:

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

Output:

```
D:\Java>javac Test.java
D:\Java>
```

- If we are performing initialization anywhere else we will get compile time error.

Example:

```
class Test
{
final int i;
public void methodOne(){
i=10;
}}
```

Output:

```
Compile time error.
D:\Java>javac Test.java
Test.java:5: cannot assign a value to final variable i
i=10;
```

Final static variables:

- If the value of a variable is not varied from object to object such type of variables is not recommended to declare as the instance variables. **We have to declare those variables at class level by using static modifier.**
- For the static variables it is not required to perform initialization explicitly jvm will always provide default values.

Example:

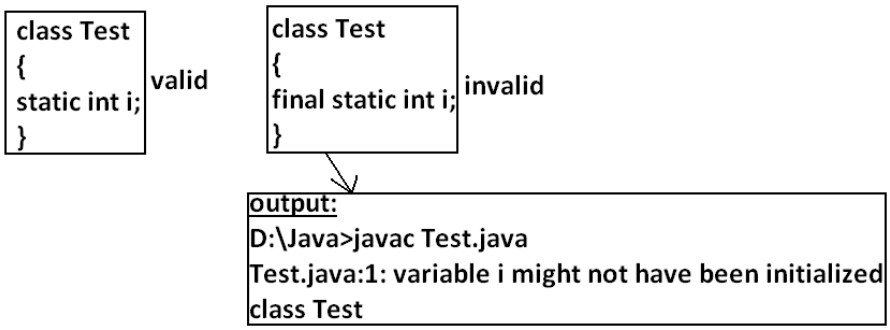
```
class Test
{
static int i;
public static void main(String args[]){
System.out.println("value of i is :"+i);
}}
```

Output:

```
D:\Java>javac Test.java
D:\Java>java Test
Value of i is: 0
```

- If the static variable declare as final then compulsory we should perform initialization explicitly whether we are using or not otherwise we will get compile time error.

Example:



Rule:

- For the final static variables we should perform initialization before class loading completion otherwise we will get compile time error. That is the following are possible places.

1) At the time of declaration:

Example:

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

Output:

```
D:\Java>javac Test.java
D:\Java>
```

2) Inside static block:

Example:

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

Output:

Compile successfully.

- If we are performing initialization anywhere else we will get compile time error.

Example:

```
class Test
{
final static int i;
public static void main(String args[]){
i=10;
```

}}

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:5: cannot assign a value to final variable i

i=10;

Final local variables:

- To meet temporary requirement of the programmer sometime we can declare the variable inside a method or block or constructor such type of variables are called local variables.
- For the local variables jvm won't provide any default value compulsory we should perform initialization explicitly before using that variable.

Example:

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

Output:

D:\Java>javac Test.java

D:\Java>java Test

Hello

Example:

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

Output:

Compile time error.

D:\Java>javac Test.java

Test.java:5: variable i might not have been initialized

System.out.println(i);

- Even though local variable declared as the final before using only we should perform initialization.

Example:

```
class Test
{
public static void main(String args[]){
final int i;
System.out.println("hello");
}}
```

Output:

```
D:\Java>javac Test.java
D:\Java>java Test
hello
```

Note: The only applicable modifier for local variables is final if we are using any other modifier we will get compile time error.

Example:

```
class Test
{
public static void main(String args[])
{
private int x=10;————(invalid)
public int x=10; ————(invalid)
volatile int x=10;————(invalid)
transient int x=10; —— (invalid)
final int x=10;————(valid)
}
}
```

Output:

```
Compile time error.
D:\Java>javac Test.java
Test.java:5: illegal start of expression
private int x=10;
```

Formal parameters:

- The formal parameters of a method are simply access local variables of that method hence it is possible to declare formal parameters as final.
- If we declare formal parameters as final then we can't change its value within the method.

Example:

```
class Test{
public static void main(String args[]){
methodOne(10,20);
}
public static void methodOne(final int x,int y){
//x=100;
y=200;
System.out.println(x+"...."+y);
}
}
```

Formal parameters

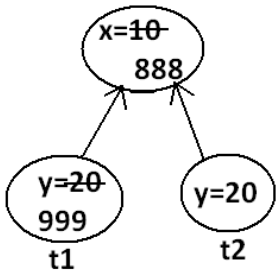
```
output:
compile time error.
D:\Java>javac Test.java
Test.java:6: final parameter x may not be assigned
x=100;
```

Static modifier:

- Static is the modifier applicable for methods, variables and blocks.
- We can't declare a class with static **but inner classes can be declaring as the static**.
- In the case of instance variables for every object a separate copy will be created but in the case of static variables a single copy will be created at class level and shared by all objects of that class.

Example:

```
class Test{
static int x=10;
int y=20;
public static void main(String args[]){
Test t1=new Test();
t1.x=888;
t1.y=999;
Test t2=new Test();
System.out.println(t2.x+"....."+t2.y);
}
}
```



Output:

D:\Java>javac Test.java

D:\Java>java Test

888.....20

- Instance variables can be accessed only from **instance area directly and we can't access from static area directly**.
- But static variables can be accessed from **both instance and static areas directly**.

1) Int x=10;
2) Static int x=10;
3) Public void methodOne(){

System.out.println(x);
}

4) Public static void methodOne(){
System.out.println(x);
}

Which are the following declarations are allow within the same class simultaneously?

a) 1 and 3

Example:

```
class Test
{
int x=10;
public void methodOne(){
System.out.println(x);
}}
```

Output:

Compile successfully.

b) 1 and 4

Example:

```
class Test
{
int x=10;
public static void methodOne(){
System.out.println(x);
}}
```

Output:

Compile time error.

```
D:\Java>javac Test.java
Test.java:5: non-static variable x cannot be referenced from a static context
System.out.println(x);
```

c) 2 and 3

Example:

```
class Test
{
static int x=10;
public void methodOne(){
System.out.println(x);
}}
```

Output:

Compile successfully.

d) 2 and 4

Example:

```
class Test
{
static int x=10;
public static void methodOne(){
System.out.println(x);
}}
```

Output:

Compile successfully.

e) 1 and 2

Example:

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

Output:

Compile time error.

```
D:\Java>javac Test.java
Test.java:4: x is already defined in Test
static int x=10;
```

f) 3 and 4

Example:

```
class Test{
public void methodOne(){
System.out.println(x);
}
public static void methodOne(){
System.out.println(x);
}}
```

Output:

Compile time error.

```
D:\Java>javac Test.java
Test.java:5: methodOne() is already defined in Test
public static void methodOne(){
```

- Overloading concept is applicable for static method including main method also.

Example:

```
class Test{
public static void main(String args[]){
System.out.println("String() method is called");
}
public static void main(int args[]){
System.out.println("int() method is called");
}
}
```
This method we have to call explicitly.

- 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.

Example:

```
class Parent{
public static void main(String args[]){
System.out.println("parent main() method called");
}
}
class child extends Parent{
}
```

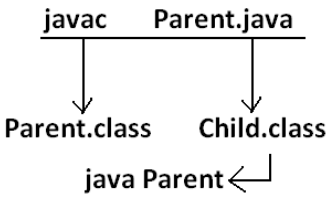
Output:

```
javac Parent.java
Parent.class  Child.class
java Parent
D:\Java>java Parent
parent main() method called
D:\Java>java child
parent main() method called
```

Example:

```
class Parent{
public static void main(String args[]){
System.out.println("parent main() method called");
}
}
class child extends Parent{
public static void main(String args[]){
System.out.println("child main() method called");
}
```
it is not overriding but method hiding.

Output:




```
D:\Java>java Parent
parent main() method called
D:\Java>java child
child main() method called
```

- It seems to be overriding concept is applicable for static methods but it is not overriding it is method hiding.
- For static methods compulsory implementation should be available where as for abstract methods implementation should be available **hence abstract static combination is illegal for methods.**

Native modifier:

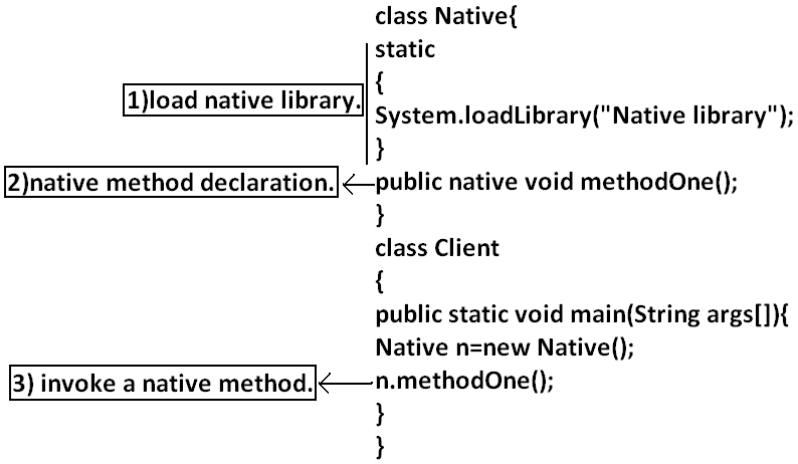
- Native is a modifier applicable only for methods but not for variables and classes.
- The methods which are implemented in non java are called native methods or foreign methods.

The main objectives of native keyword are:

- To improve performance of the system.
- To use already existing legacy non java code.

To use native keyword:

Pseudo code:



- For native methods implementation is already available and we are not responsible to provide implementation hence native method declaration should compulsory ends with semicolon.
    - Public native void methodOne()----invalid
    - Public native void methodOne();---valid
- For native methods implementation is already available where as for abstract methods implementation should not be available child class is responsible to provide that, hence abstract native combination is illegal for methods.

- We can't declare a native method as strictfp because there is no guaranty whether the old language supports IEEE754 standard or not. That is native strictfp combination is illegal for methods.
- For native methods inheritance, overriding and overloading concepts are applicable.
- The main disadvantage of native keyword is usage of native keyword in java breaks platform independent nature of java language.

Synchronized:

- Synchronized is the modifier applicable for methods and blocks but not for variables and classes.
- If a method or block declared with synchronized keyword then at a time only one thread is allow to execute that method or block on the given object.
- The main advantage of synchronized keyword is we can resolve data inconsistency problems, but the main disadvantage is it increases waiting time of the threads and effects performance of the system. Hence if there is no specific requirement never recommended to use synchronized keyword.

Transient modifier:

- Transient is the modifier applicable only for variables but not for methods and classes.
- At the time of serialization if we don't want to serialize the value of a particular variable to meet the security constraints then we should declare that variable with transient modifier.
- At the time of serialization jvm ignores the original value of the transient variable and save default value that is transient means "not to serialize".
- Static variables are not part of object state hence serialization concept is not applicable for static variables duo to this declaring a static variable as transient there is no use.
- Final variables will be participated into serialization directly by their values due to this declaring a final variable as transient there is no impact.

Volatile modifier:

- Volatile is the modifier applicable only for variables but not for classes and methods.
- If the value of variable keeps on changing such type of variables we have to declare with volatile modifier.
- If a variable declared as volatile then for every thread a separate local copy will be created by the jvm, all intermediate modifications performed by the thread will takes place in the local copy instead of master copy.
- Once the value got finalized before terminating the thread that final value will be updated in master copy.
- The main advantage of volatile modifier is we can resolve data inconsistency problems, but creating and maintaining a separate copy for every thread increases complexity of

the programming and effects performance of the system. Hence if there is no specific requirement never recommended to use volatile modifier and it's almost outdated.

- Volatile means the value keep on changing where as final means the value never changes hence final volatile combination is illegal for variables.

Modifier	Classes		Methods	Variables	Blocks	Interfaces	Enum	Constructors
	Outer	Inner						
Public	✓	✓	✓	✓	✗	✓	✓	✓
Private	✗	✓	✓	✓	✗	✗	✗	✓
Protected	✗	✓	✓	✓	✗	✗	✗	✓
Default	✓	✓	✓	✓	✗	✓	✓	✓
Final	✓	✓	✓	✓	✗	✗	✗	✗
Abstract	✓	✓	✓	✗	✗	✓	✗	✗
Strictfp	✓	✓	✓	✗	✗	✓	✓	✗
Static	✗	✓	✓	✓	✓	✗	✗	✗
Synchronized	✗	✗	✓	✗	✓	✗	✗	✗
Native	✗	✗	✓	✗	✗	✗	✗	✗
Transient	✗	✗	✗	✓	✗	✗	✗	✗
Volatile	✗	✗	✗	✓	✗	✗	✗	✗

Summary of modifier:

- The modifiers which are applicable for inner classes but not for outer classes are **private, protected, static**.
- The modifiers which are applicable only for methods **native**.
- The modifiers which are applicable only for variables **transient and volatile**.
- The modifiers which are applicable for constructor public, private, protected, default.
- The only applicable modifier for local variables is **final**.

Interfaces:

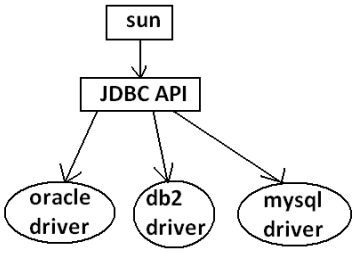
1) Introduction
2) Interface declarations and implementations.
3) Extends vs implements
4) Interface methods
5) Interface variables
6) Interface naming conflicts
   a) Method naming conflicts
   b) Variable naming conflicts
7) Marker interface
8) Adapter class
9) Interface vs abstract class vs concrete class.
10) Difference between interface and abstract class?

11) Conclusions

Def1: Any service requirement specification (srs) is called an interface.

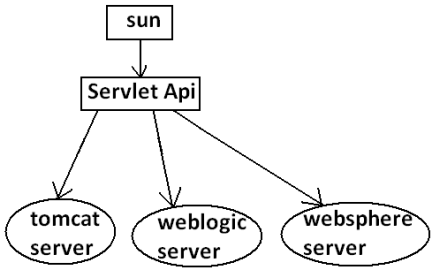
Example1: Sun people responsible to define JDBC API and database vendor will provide implementation for that.

Diagram:



Example2: Sun people define SERVLET API to develop web applications web server vendor is responsible to provide implementation.

Diagram:



Def2: From the client point of view an interface define the set of services what his excepting. From the service provider point of view an interface defines the set of services what is offering. Hence an interface is considered as a contract between client and service provider.

Example: ATM GUI screen describes the set of services what bank people offering, at the same time the same GUI screen the set of services what customer his excepting hence this GUI screen acts as a contract between bank and customer.

Def3: Inside interface every method is always abstract whether we are declaring or not hence interface is considered as 100% pure abstract class.

Summery def: Any service requirement specification (SRS) or any contract between client and service provider or 100% pure abstract classes is considered as an interface.

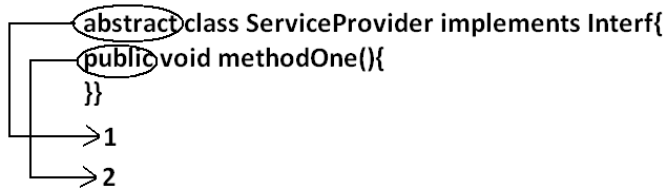
Declaration and implementation of an interface:

Note1: Whenever we are implementing an interface **compulsory for every method of that interface we should provide implementation otherwise we have to declare class as abstract** in that case **child class is responsible to provide implementation for remaining methods**.

Note2: Whenever we are implementing an interface method **compulsory it should be declared as public otherwise we will get compile time error**.

Example:

```
interface Interf
{
void methodOne();
void methodTwo();
}
```




```
class SubServiceProvider extends ServiceProvider
{
}
```

Output:

Compile time error.

```
D:\Java>javac SubServiceProvider.java
SubServiceProvider.java:1: SubServiceProvider is not abstract and does not override abstract method methodTwo() in Interf
class SubServiceProvider extends ServiceProvider
```

Extends vs implements:

- A class can extend only one class at a time.

Example:

```
class One{
public void methodOne(){
}
}
class Two extends One{
}
```

- A class can implements any no. Of interfaces at a time.

Example:

```
interface One{
public void methodOne();
}
interface Two{
public void methodTwo();
}
class Three implements One,Two{
public void methodOne(){
}
```

```
public void methodTwo(){
}
}
```

- A class can extend a class and can implement an interface simultaneously.

```
interface One{
void methodOne();
}
class Two
{
public void methodTwo(){
}
}
class Three extends Two implements One{
public void methodOne(){
}
}
```

- An interface can extend any no. Of interfaces at a time.

Example:

```
interface One{
void methodOne();
}
interface Two{
void methodTwo();
}
interface Three extends One,Two
{
}
```

1) **Which of the following is true?**
    1. A class can extend any no. Of classes at a time.
    2. An interface can extend only one interface at a time.
    3. A class can implement only one interface at a time.
    4. A class can extend a class and can implement an interface but not both simultaneously.
    5. None of the above.

Ans: 5

2) Consider the expression **X extends Y** for which of the possibility of X and Y this expression is true?
    1. Both x and y should be classes.

2. Both x and y should be interfaces.
3. Both x and y can be classes or can be interfaces.
4. No restriction.

Ans: 3

3) X extends Y, Z?

- X, Y, Z should be interfaces.

4) X extends Y implements Z?

- X, Y should be classes.
- Z should be interface.

5) X implements Y, Z?

- X should be class.
- Y, Z should be interfaces.

6) X implements Y extend Z?

Example:

```
interface One{
}
class Two {
}
class Three implements One extends Two{
}
```

Output:

```
Compile time error.
D:\Java>javac Three.java
Three.java:5: '{' expected
class Three implements One extends Two{
```

- Every method present inside interface is always **public and abstract** whether we are declaring or not. Hence inside interface the following method declarations are equal.

```
void methodOne();
public Void methodOne();
abstract Void methodOne();
public abstract Void methodOne();
```

Equal

- As every interface method is always public and abstract we can't use the following modifiers for interface methods.
- **Private, protected, final, static, synchronized, native, strictfp.**

Inside interface which method declarations are valid?

1. public void methodOne(){}
2. private void methodOne();
3. public final void methodOne();

4. public static void methodOne();
5. public abstract void methodOne();

Ans: 5

Interface variables:

- An interface can contain variables to define requirement level constants.
- Every interface variable is always **public static and final** whether we are declaring or not.

Example:

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

Public: To make it available for every implementation class.

Static: Without existing object also we have to access this variable.

Final: Implementation class can access this value but cannot modify.

- Hence inside interface the following declarations are equal.

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

Equal

- As every interface variable by default **public static final** we can't declare with the following modifiers.
    - Private
    - Protected
    - Transient
    - Volatile
- For the interface variables compulsory we should perform initialization at the time of declaration only otherwise we will get compile time error.

Example:

```
interface Interf
{
int x;
}
```

Output:

Compile time error.

D:\Java>javac Interf.java

Interf.java:3: = expected

int x;

Which of the following declarations are valid inside interface?

1. int x;
2. private int x=10;
3. public volatile int x=10;
4. public transient int x=10;
5. public static final int x=10;

Ans: 5

- Interface variables can be access from implementation class but cannot be modified.

Example:

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

Example 1:

```
class Test implements Interf
{
public static void main(String args[]){
x=20;
System.out.println("value of x"+x);
}
}
```

output:

compile time error.

```
D:\Java>javac Test.java
Test.java:4: cannot assign a value to final variable x
x=20;
```

Example 2:

```
class Test implements Interf
{
public static void main(String args[]){
int x=20;
//here we declaring the variable x.
System.out.println(x);
}
}
```

Output:

```
D:\Java>javac Test.java
D:\Java>java Test
20
```

Interface naming conflicts:

Method naming conflicts:

Case 1:

- If two interfaces contain a method with same signature and same return type in the implementation class only one method implementation is enough.

Example 1:

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

Example 2:

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

Example 3:

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

Output:

```
D:\Java>javac Left.java
D:\Java>javac Right.java
D:\Java>javac Test.java
```

Case 2:

- if two interfaces contain a method with same name but different arguments in the implementation class we have to provide implementation for both methods and these methods acts as a overloaded methods

Example 1:

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

Example 2:

```
interface Right
{
public void methodOne(int i);
}
```

Example 3:

```
class Test implements Left,Right
{
public void methodOne()
{
}
public void methodOne(int i)
{
}}
```

Output:

```
D:\Java>javac Left.java
D:\Java>javac Right.java
D:\Java>javac Test.java
```

Case 3:

- If two interfaces contain a method with same signature but different return types then it is not possible to implement both interfaces simultaneously.

Example 1:

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

Example 2:

```
interface Right
{
public int methodOne(int i);
}
```

- We can't write any java class that implements both interfaces simultaneously.

Is 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.

Variable naming conflicts:

- Two interfaces can contain a variable with the same name and there may be a chance variable naming conflicts but we can resolve variable naming conflicts by using interface names.

Example 1:

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

Example 2:

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

Example 3:

```
class Test implements Left,Right
{
public static void main(String args[]){
//System.out.println(x);
System.out.println(Left.x);
System.out.println(Right.x);
}
}
```

Output:

```
D:\Java>javac Left.java
D:\Java>javac Right.java
D:\Java>javac Test.java
D:\Java>java Test
888
999
```

Marker interface: if an interface doesn't contain any methods and by implementing that interface if our object gets some ability such type of interfaces are called Marker interface (or) Tag interface (or) Ability interface.

Example:

Serilizable
cloneable
RandomAccess
SingleThreadModel
} These are marked for some ability

.
.
.
.

Example 1: By implementing Serilizable interface we can send that object across the network and we can save state of an object into a file.

Example 2: By implementing SingleThreadModel interface Servlet can process only one client request at a time so that we can get "Thread Safety".

Example 3: By implementing Cloneable interface our object is in a position to provide exactly duplicate cloned object.

Without having any methods in marker interface how objects will get ability?

- Internally JVM will provide required ability.

Why JVM is providing the required ability?

- To reduce complexity of the programming.

Is it possible to create our own marker interface?

- Yes, but customization of JVM is required.

Adapter class:

- Adapter class is a simple java class that implements an interface only with empty implementation for every method.
- If we implement an interface directly for each and every method compulsory we should provide implementation whether it is required or not. This approach increases length of the code and reduces readability.

Example 1:

```
interface X{
void m1();
void m2();
void m3();
void m4();
 //.
 //.
 //.
 //.
void m5();
}
```

Example 2:

```
class Test implements X{
public void m3(){
System.out.println("m3() method is called");
}
public void m1(){}
public void m2(){}
public void m4(){}
```

```
public void m5(){}
}
```

- We can resolve this problem by using adapter class.
- Instead of implementing an interface if we can extend adapter class we have to provide implementation only for required methods but not for all methods of that interface.
- This approach **decreases length of the code** and improves readability.

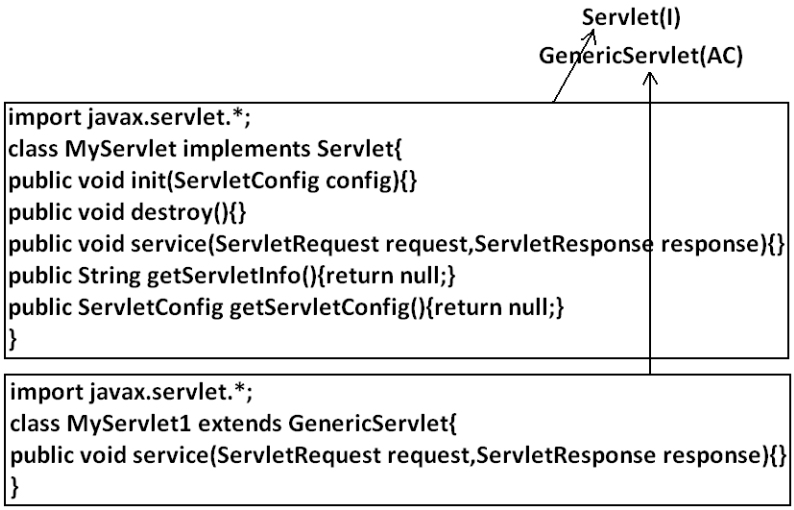
Example 1:

```
abstract class AdapterX implements X{
public void m1(){}
public void m2(){}
public void m3(){}
public void m4(){}
 //.
 //.
 //.
public void m1000(){}
}
```

Example 2:

```
public class Test extend AdapterX{{
public void m3(){
}}
```

Example:



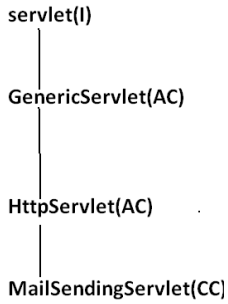
- Generic Servlet simply acts as an adapter class for Servlet interface.

What is the difference between interface, abstract class and concrete class?

When we should go for interface, abstract class and concrete class?

- If we don't know anything about implementation just we have requirement specification then we should go for interface.
- If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.
- If we are talking about implementation completely and ready to provide service then we should go for concrete class.

Example:



What is the Difference between interface and abstract class?

interface	Abstract class
1) If we don't' know anything about implementation just we have requirement specification then we should go for interface.	1) If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.
2) Every method present inside interface is always **public and abstract** whether we are declaring or not.	2) Every method present inside abstract class **need not be public and abstract**.
3) We can't declare interface methods with the modifiers **private**, **protected**, **final**, **static**, **synchronized**, **native**, **strictfp**.	3) There are no restrictions on abstract class method modifiers.
4) Every interface variable is always **public static final** whether we are declaring or not.	4) Every abstract class variable need not be public static final.
5) Every interface variable is always **public static final** we can't declare with the following modifiers. **Private, protected, transient, volatile.**	5) There are no restrictions on abstract class variable modifiers.
6) For the interface variables compulsory we should perform initialization at the time of declaration otherwise we will get compile time error.	6) It is not require to perform initialization for abstract class variables at the time of declaration.
7) Inside interface we can't take static and instance blocks.	7) Inside abstract class we can take both static and instance blocks.

8) Inside interface we can't take constructor.	8) Inside abstract class we can take constructor.

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

- This constructor will be executed for the initialization of child object.

Example:

```
class Parent{
Parent()
{
System.out.println(this.hashCode());
}
}
class child extends Parent{
child(){
System.out.println(this.hashCode());
}
}
class Test{
public static void main(String args[]){
child c=new child();
System.out.println(c.hashCode());
}
}
```

Every method present inside interface is abstract but in abstract class also we can take only abstract methods then what is the need of interface concept?

- We can replace interface concept with abstract class. But it is not a good programming practice. We are misusing the roll of abstract class.