Declaration and Access Modifiers

Java source file structure

- A java program can contain any number of classes but at most 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 can give for java source file

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

```
class A
{
        public static void main(String …s)
        {
                System.out.println(“class A”)
}
}

class B
{
        public static void main(String …s)
        {
                System.out.println(“class B”)
}

}
class C
{
        public static void main(String …s)
        {
                System.out.println(“class A”)
}
}
class D
{

}
```

After compiling this there will be creation of four .class file like A.class, B.class, C.class and D.class. In java we can run any .class file for output but not .java file. So we can have output like:

output

java A

class A

java B

class B

java C

C

java D

Run time exception: moclassdeffound.

It is not recommended to declare multiple classes in a single source file. It is recommended to declare only one class per source file and name of the program we have to keep same as class name. The main advantage of this approach is readability and maintainability of the code will be improved.

Import statement

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

Output

Compile time error.
cannot find symbol
Arraylist l=new ArrayList();
symbol: class Arraylist
location: class Test
Arraylist l=new Arraylist();
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();
        }
}
```

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.

Types of Import Statements

There are two types of import statements.

Explicit class import

Example

import java.util.Arraylist

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

Implicit class import:

Example

import java.util. *;

It is never recommended to use because it reduces readability of the code.

Whenever we are importing a package all classes and interfaces present in that package are by default available but not sub package classes. If we want use sub package class compulsory we should write import statement until sub package level

Java→util→regex→Pattern
import java.util.regex. *;
import java.util.regex.Pattern;
to use pattern class in our program above import statements are valid.
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)

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 static imports creates confusion and reduces readability of the code. Hence if there is no specific requirement never recommended to use a static import. 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(l0,20));
		System.out.println(Math.random());
	}
}
```

Output

```
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.ot.:t.println(sqrt(4));
```

```
                System.out.println( max( 10,20) );
                System.out.println{random());
        }
}

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

Output

Compile time error.
reference to MAX_VALUE is ambiguous, both variable MAX_VALUE in java.lang.lnteger and variable MAX_VALUE in java.lang.Byte match System.out.println(MAX_ VALUE);

Two packages contain a class or interface with the same is very rare hence ambiguity problem is very rare in normal import. But two 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 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 static java.lang.lnteger.          MAX_ VALUE;        →line2
import static java.lang.Byte. *;
class Test
{
        //static int MAX_VALUE=999;                                  →line 1
        public static void main(String args[])throws Exception
        {
                System.out.println(MAX_ VALUE);
        }
}
```

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

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

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

1. All classes and interfaces which are required for database operations are grouped into a single package which is nothing but java.sql package

2. All classes and interfaces which are usefull for file I/O operations are grouped into a separate package which is nothing but java.io package.

The main objectives of packages are

- To resolve name conflicts (unique identification of our components).
- To improve modularity of the application.
- To provide security

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

Javac -d . job.java

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

Or

Example

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

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

How to execute package program:

D:\Java>java com.pack

- At the time of execution compulsory, we should provide fully qualified name.
- 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.
- In any java program the first non-comment statement should be package statement [if it is available] otherwise we will get compile time error.

Java sourse file structure

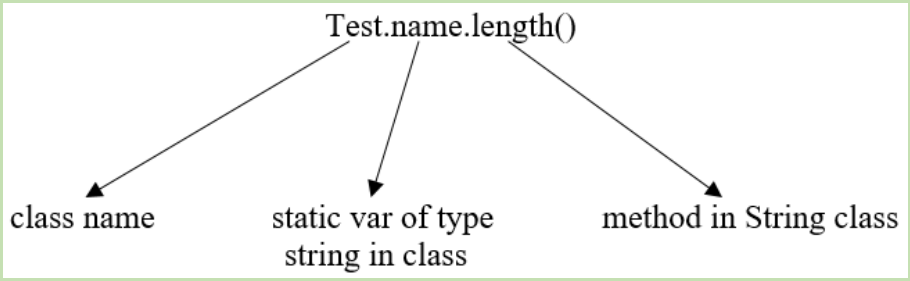
Package statement	Atmost one allowed
Import statements	any number is allowed
Class/interface/enum declaration	any number is allowed

All the following are valid java programs.

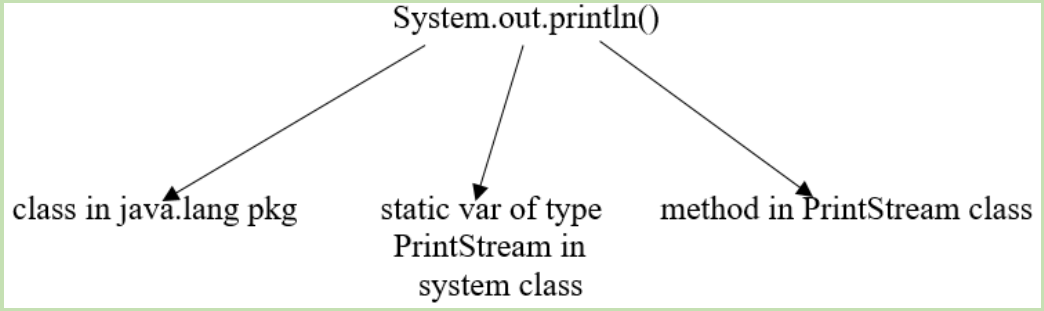
	package pack1;	**import java.util.*;**	**package pack1;** **import java.util.*;**	**class Test** **{** **}**
Test.java	**Test.java**	**Test.java**	**Test.java**	**Test.java**

System.out.println statement

```
class Test
{
        static String name="bhaskar"
}
```




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




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

Class Modifiers

Whenever we are writing our own classes compulsory, we have to provide some information about our class to the jvm. like

- Whether class can be accessible from anywhere or not.
- Whether child class creation is possible or not.
- 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=O;
                for(int j=O;j<3;j++)
                {
                        i=i+j;
                }
                System.out.println(i);
}
}
```

Output

Compile time error.
modifier private not allowed here
private class Test

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

Public	private
Default	protected
Final	static
Abstract	Strictfp

The difference between access specifier and access modifier

In '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 is declared as public then we can access that class from anywhere.

Example

Program 1

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

Compile the above program:
D:\Java>javac -d . Test.java

Program2:

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

Output
Test class methadone is executed.

If class Test is not public then while compiling Testl class we will get compile time error saying packl.Test is not public in packl; 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".

```
package pck;
class defpack
{
        public void method()
        {
                System.out.println("default modifier");
        }
}
```

Javac – d. defpack.java

```
package pck;
import pk.defpack;
class defpackk
{
        public static void main(String[] args)
        {
                defpack d=new defpack();
                d.method();
        }
}
```

Output
default modifier

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

Final Methods

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

Example

```
class Parent
{
        public void fn()
        {
                System.out.println("function 1"};
        }
        public final void fn2()
        {
                System.out.printl n("function 2");
        }
}
```

```
class child extends Parent
{
        public void fn2()
        {
                System.out.println("Thamanna");
        }
}
```

Compile time error.
Fn2() in child cannot override fn2() in Parent; overridden method is final
public void fn2()

Final Class

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

Example

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

Output

Compile time error.
cannot inherit from final Parent
class child extends Parent

- 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.

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 end with semicolon.

Example

```
public abstract void methadone();          valid
public abstract void methodOne(){}         invalid
```

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

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

```
class Bus extends vehicle
{
        public int getNoOfWheels()
```

```
class Auto extends vehicle
{
        public int getNoOfWheels()
```

```
    {
        return 7;
    }
}
```

```
    {
        return 3;
    }
}
```

- 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 compulsorily 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.

	final,
	native,
	strictfp,
abstract	
	static,
	private,
	synchronized

Abstract class

For any java class if we are not allowed to create an object then, 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.
Test is abstract; cannot be instantiated
Test t=new Test();
```

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 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.

If a class extends any abstract class then it is mandatory 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(){}
}
```

Compile time error.
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().

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 must 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.

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. 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.

System.out.println(l0.0/3);

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.

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

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

Program 2:

```
package pack2;
import packl.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: packl.A is not public in packl; cannot be accessed from outside package
import packl.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

Program 1

```
package packl;
class A
{
        void methodOne()
        {
                System.out.println( 11methodOne is executed");
        }
}
```

Program 2

```
package packl;
import packl.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 packl.B
methodOne is executed
```

Example

Program 1

```
package packl;
class A
{
        void methodOne()
```

```
	{
		System.out.println("methodOne is executed");
	}
}
```

Program 2

```
package pack2;
import packl.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: packl.A is not public in packl; cannot be accessed from outside package
import packl.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 whereas 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 packl;
public class A
{
	protected void methodOne()
	{
		System.out.println{"methodOne is executed");
	}
}
```

Program 2

```
package packl;
class B extends A
{
	public static void main(String args[])
	{
		A a=new A();
```

```
                a.methodOne();
                B b=new B();
                b.methodOne();
                A al=new B();
                al.methodOne();
        }
}
```

Output

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

```
package pack2;
import packl.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 al=new B(};
                al.methodOne();
        }
}
```

output

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

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. 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

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

```
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

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

- **At the time of declaration:**

Example

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

- **Inside instance block:**

Example

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

- **Inside constructor**

Example

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

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

If we perform 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.

For the final static variables, we should perform initialization before class loading completion otherwise, we will get compile time error. These are the following 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

Hello

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

Output

```
Compile time error.
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

Hello

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

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.

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=l0;
	int y=20;
	public static void main (String args[])
	{
		Test tl=new Test();
		tl.x=888;
		tl.y=999;
		Test t2=new Test();
		System.out.println(t2.x+" ..... "+t2.y);
	}
}
```

Output

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.

```
int x=l0;
static int x=l0;
public void methodOne()
{
	System.out.println(x);
}
public static void methodOne()
{
	System.out.println(x);
}
```

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

- 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 whereas final means the value never changes hence final volatile combination is illegal for variables.