Declarations & Access Modifiers

Topics

1. Java Source File Structures – Import, static Import, Package,
2. Class level Modifiers – public, default, final, abstract, strictfp
3. Member level Modifiers – public, default, private, protected, final, static, synchronized, native, transient, volatile
4. Interfaces – Marker interface, Adapter interface

1. Java Source File Structures

- A Java program can contain any no. of classes but at most one class can be declared as public.
- If there is a public class, then name of the program & name of the public class must be matched otherwise we will get compile time error.
  - **Case 1:** - If there is no public class then we can use any name & there are no restrictions like A.java, B.java, C.java etc.
  - **Case 2:** - If Class B is public then name of the program should B.java otherwise we will get compile time error.
  - **Case 3:** - If Class B & C declared as public & name of the program is B.java then we will get compile time error.

- Whenever we're compiling a java program, for every class present in that program, a separate .class file will be generated.
- We can compile a java program (Java source file) but we can run only a java .class file.
- Whenever we're executing a java class, the corresponding class main method will be executed. If the class doesn't contain main () method then we will get Runtime Exception: **NoSuchMethodError**: main
- If corresponding .class file not available then we will get Runtime Exception: **NoClassDefFoundError**
- It is recommended to declare only one class per single source file & keep name of program same as class name. The advantage of this approach is readability & maintainability of the code will be improved.

Import Statement

- For using a class in Java, we need to import the required classes into our class
  - One way is to use fully Qualified Name. But the problem with fully qualified name is it increases the length of the code & reduces readability.
  - Another way is using "import" statement. When using "import" statement, we don't need to write fully qualified name of the required class, just use short name directly.

- Types of Import statements
  a) Explicit class import
  b) Implicit class import

a) Explicit class import

- It is highly recommended to use explicit class import because it improves readability of the code.
  e.g. import java.util.ArrayList;

b) Implicit class import

- It is not recommended to use because it reduces readability of the code.
  e.g. import java.util.*;

Notes:

- While resolving class names, compiler will always give precedence in the following order for normal import not static import.
    a) 1. Explicit class import //Highest priority
    b) 2. Classes present in Current working directory (CWD) or default package.
    c) 3. Implicit class import // Lowest priority

- Whenever we're importing a java package, all classes & interfaces present in that package by default available but not sub-package class, compulsory we should write import statement until sub-package level.

- All classes & interfaces present in the following packages are by default available to every java program; hence we're not required to write import statement
    a) java.lang package
    b) default package i.e. current working directory

- Import statements is totally compile time related concept. If more no. of imports then more will be the compile time but there is no affect on execution time (runtime).

Q. Difference between C language #include & java language import statement?
Ans: In case of C language #include, all input, output, header files will be loaded at the beginning only (at translation time); hence it is **static include**.
But In case of Java import statement, no .class file will be loaded at the beginning. Whenever we require a particular class then only corresponding .class file will be loaded. This is like **dynamic include** or **Load on Demand** or **Load on Fly**.

Static Import

- If there is no specific requirement then it is not recommended to use static import as it creates confusion & reduces readability.
- Usually, we can access static members by using class name but whenever we're writing static import, we can access static members directly without class name.

e.g.

Without static import	With static import

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

Q. Explain about **System.out.println**

Ans: - class
```
System {
        Static PrintStream out;
        ..
        ..
}
```

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

"**out**" is a static variable present in "**System**" class hence to access out we need to use class name "System". But whenever we're writing static import, it is not required to use class name & we can access "out" directly.

Without static import	With static import

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

- While resolving static members, compiler will always consider the precedence in the following order
  1) Current class static members // Highest Priority
  2) Explicit static import
  3) Implicit static import // Lowest Priority

Normal import Vs Static import

No	Normal import	Static import
1.	Explicit import Syntax: import *package_name*.**class_name;** e.g. import *java.util*.**ArrayList**;	Explicit static import Syntax: import static *package_name*.**class_name**.static_member**;** e.g. import static *java.lang*.**Math**.sqrt;
2.	Implicit import Syntax: import *package_name*.***;** e.g. import *java.util*.*****;	Implicit static import Syntax: import static *package_name*.**class_name**.***;** e.g. import static *java.lang*.**Math**.*;

Problems with Static import

- 2 packages contain a class or interface with same name is very rare & hence ambiguity problem is also very rare in normal import. But 2 classes or interfaces contain a variable or method with same name is very common & hence ambiguity problem is also very common problem in static import.
- Usage of static import reduces readability & creates confusion & hence if there is no specific requirement then it is not recommended to use static import.

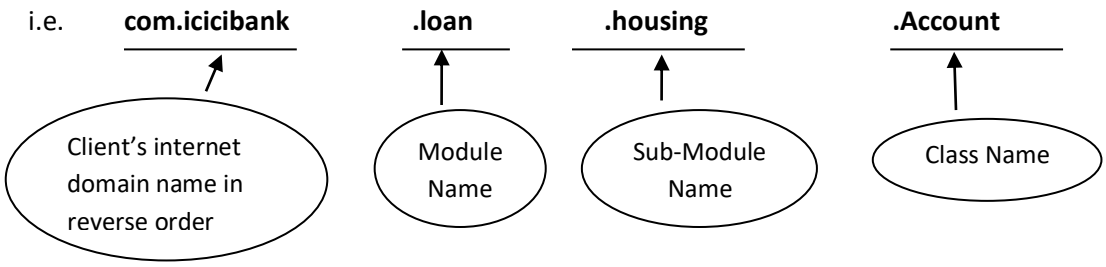
Packages

- It is an encapsulation mechanism to group related classes & interfaces into a single unit, which is nothing but package.

  e.g. All classes & interfaces which are useful for **file operations** are grouped in a separate package which is nothing but **java.io** package.

- Advantages of packages are
  a) To resolve naming conflicts (i.e. unique identification of our component)
  b) It improves modularity of the application.
  c) It improves maintainability of the application.
  d) It provides security of our component.

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



- In any java source file, there can only be atmost one package statement.

- In any java program, the first non-comment statement should be package statement.

- The following is valid Java source file structure (Order of statements is important). Also, any empty source file is a valid java program.

```
package statement;    // at most one

import statement;     // any number

class/interface/enum declaration  // any number
```

2. Class level Modifiers

- Whenever we're writing our own classes, we have to provide some information about our class to the JVM like
  a) Whether this class can be accessible from anywhere or not.
  b) Whether child class creation is possible or not
  c) Whether object creation is possible or not. etc. we can specify all these information by using appropriate modifier.

- **Access Specifier Vs Access Modifier:** There is no word like Specifier in Java but in old languages, private, protected, public, default are considered as Access specifiers.

- The only possible modifiers for top – level classes & Inner classes are

Top level classes	Inner Classes
1. public 2. <default> 3. final 4. abstract 5. strictfp	1. public 2. <default> 3. final 4. abstract 5. strictfp 6. private 7. protected 8. static

1. public Classes: If a class is declared as public then we can access that class from anywhere.

2. default Classes: If a class declared as default then we can access that class only within the current package i.e. from outside package we can't access, hence default class is also known as package level access. There is nothing such private classes or protected classes.

3. final Modifier

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

a) final Method

- Whatever methods parent has by default available to the child through inheritance. If the child is not satisfied with parent method implementation, then child is allowed to redefine that method based on its requirements. This process is called **Overriding**.
- So, if a parent class method is declared as final, then the child class can't override that method because its implementation is final.

b) final class

- If a class is declared as final, we can't extend functionality of that class i.e. we can't create child class for that class i.e. Inheritance is not possible for final classes.
- Every method present inside final class is always final by default but every variable present inside final class need not to be final.

- The main **advantage** of final keyword is we can achieve security & we can provide unique implementation.
- But the main **disadvantage** of final keyword is we're missing key benefits of OOPs: inheritance (because of final classes) & Polymorphism (because of final methods). Hence if there is no specific requirement then it is not recommended to use final keyword.

4. abstract Modifier

- abstract is the modifier applicable for classes & methods only not to variables.

a) Abstract method

- Even though we don't know about implementation still we can declare a method with abstract modifier i.e. for abstract methods only declaration is available but not implementation.

```
e.g.    public abstract void m1 ();        // correct
        public abstract void m1 () { }     // wrong
```

- Abstract method never talks about implementation if any modifier talks about implementation, then it forms illegal combination with abstract modifier.

b) Abstract class

- For any java class, if we are not allowed to create an object (because of partial implementation), such type of class should be declared with abstract modifier i.e. for abstract classes, instantiation is not possible.

Note:

- If a class contains at least one abstract method then compulsory we should declare class as abstract otherwise we will get compile time error.
- Even though a class doesn't contain any abstract method still we can declare the class as abstract if we don't want instantiation i.e. abstract class can contain zero number of abstract methods also.
  e.g. HttpServlet class
- If a class extends an abstract class, then the child class has to provide implementation for each & every abstract method or we should declare the child class as abstract (in this case, next level child class is responsible to provide implementation).
- Abstract class can contain final method.
- It is highly recommended to use abstract modifier because it promotes several OOPs features like inheritance & polymorphism.

5. strictfp [strict floating point]

- We can declare/use strictfp for classes & methods but not for variables.
- Usually, the result of floating-point arithmetic is varied from platform to platform. If we want platform independent results for floating point arithmetic, then we should go for strictfp modifier.

a) strictfp method

- If a method declared as strictfp, all floating-point calculations in that method has to follow IEEE 754 Standard, so that we will get platform independent results.
- We can't declare abstract – strictfp combination for methods.

b) strictfp class

- If a class is declared as strictfp then every floating-point calculation present in every concrete method has to follow IEEE 754 Standard so that we will get platform independent results.
- We can declare abstract – strictfp combination for classes.

3. Member level Modifiers

1. public members

- If a member is declared as public then we can access that member from anywhere but corresponding class should be visible i.e. before checking member visibility we have to check class visibility.

2. default members

- If a member declared as default then we can access that member only within the current package i.e. from outside of the package we can't access hence default access is also known as package level access.

3. private members

- If a member is declared as private then we can access that member only within the class i.e. from outside of the class we can't access.
- private – abstract combination is illegal for methods as abstract methods should be available to the child classes to provide implementation whereas private methods are not available to the child classes.

4. protected members

- If a member is declared as protected then we can access that member anywhere within current package but only in child classes of outside package.

  i.e. **protected = default + children**

- We can access protected members within the current package anywhere either by using parent reference or child reference but we can access protected members in outside package only in child classes by using child reference only i.e. parent reference can't be used to access protected members from outside package.

No.	Visibility	private	default	protected	public
1.	Within the same class	Yes	Yes	Yes	Yes
2.	From child class of same package	No	Yes	Yes	Yes
3.	From non – child class of same package	No	Yes	Yes	Yes
4.	From child class of outside package	No	No	Yes (we should use child reference)	Yes
5.	From non – child class of outside package	No	No	No	Yes
6.	Order of restriction	Most restricted access modifier			Least/ most accessible

- **Recommended modifier for data member** (variable) is private & **recommended modifier for method** is public.

5. final variable

a) final instance variable

- If the value of a variable is varied from object to object, such type of variable is called instance variable.
- If an instance variable is declared as final then compulsory, we have to perform initialization explicitly whether we're using or not & JVM won't provide default values.
- Only places where final instance variable can be initialized:

a) At the time of declaration

e.g.
```
class Test {
	final int x = 10;
}
```

b) Inside instance block

e.g.
```
class Test {
	final int x;
	{
		x = 10;
	}
}
```

c) Inside constructor

e.g.
```
class Test {
	final int x;
	Test () {
		x = 10;
	}
}
```

b) final static variables

- If the value of a variable is not varied from object to object, such types of variables are declared as static.
- If a static variable is declared as final then compulsory, we should perform initialization explicitly otherwise we will get compile time error & JVM won't provide any default values.
- For final static variables compulsory, we should perform initialization before class loading completion.
- Only places where final instance variable can be initialized:

a) At the time of declaration

e.g.
```
class Test {
	final static int x = 10;
}
```

b) Inside static block

e.g.
```
class Test {
	final static int x;
	static {
		x = 10;
	}
}
```

c) final local variables

- For local variables, JVM won't provide any default values, compulsory we should perform initialization explicitly before using that local variable.
- If we're not declaring any modifier then by default it is default but this rule is only applicable for instance & static variables but not for local variables.
- **Formal parameters of a method**: Formal parameters of a method simply access local variables of that method; hence formal parameter can be declared as final then within the method we can't perform reassignment.

6. static modifier

- static is a modifier applicable for methods & variables but not for classes.
- We can't declare top level class with static modifier but we can declare inner class as static (such type of inner class is called static nested classes).
- In the case of instance variables, for every object a separate copy will be created but in the case of static variable, a single copy will be created at class level & shared by every object of that class.

E.g.

```
public 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("T1.x = " + t1.x + " T1.y = " + t1.y);
        System.out.println("T2.x = " + t2.x + " T2.y = " + t2.y);
     }
}

Output:
 t1.x = 888 t1.y = 999
 t2.x = 888 t2.y = 20
```

- We can't access instance member directly from static area but we can access static member directly from instance area or static area directly.

Case 1: Overloading concept applicable for static methods including main () method but JVM will always call String [] argument main () method only.

Case 2: Inheritance concept applicable for static methods including main () method, hence while executing child class if a child doesn't contain main () method then parent class main () method will be executed.

Case 3: It seems overriding concept applicable for static method but it's not overriding, its method hiding.

- abstract – static combination is illegal for methods as static method has implementation.

7. synchronized modifier

- synchronized is a modifier applicable for methods & blocks but not for classes and variables.
- If multiple threads are trying to operate simultaneously on the same java object, then there may be a chance of data inconsistency problem, this is called **race condition**, we can overcome this problem by using synchronized keyword.
- If a method or block is declared as synchronized then at a time, only one thread is allowed to execute that method or block on the given object so that data inconsistency problem will be resolved.
- But the main disadvantage of synchronized keyword is it increases waiting time of threads & creates performance problems; hence if there is no specific requirement then it is not recommended to use synchronized keyword.

- abstract – synchronized is illegal combination for methods as synchronized method has implementation.

7. native modifier

- native is the modifier applicable only for methods & we can't apply anywhere else.
- The methods which are implemented in non – java (mostly C or C++) are called native methods or foreign methods.
- The main objectives of native keyword are
  1. To improve performance of the system.
  2. To achieve machine level or memory level communication.
  3. To use already existing legacy non – java code.

- The main important benefit is wherever performance is critical & java is not up to the mark with respect to performance, we can fill this gap by using native keyword.

- abstract – native combination is illegal combination for methods as native methods implementation is already available in old language.
- strictfp – native combination is illegal for methods as there is no guarantee that old languages follow IEEE 754 Standard.
- The main advantage of native keyword is performance will be improved but the main disadvantage of native keyword is it breaks platform independent nature of java (i.e. C/C++ are platform dependent languages).
- Pseudocode to use native keyword in Java

```
class NativeClass {

     static {
         System.out.println("Native library path");
     }
     public native void loadNativeLibrary();
}

public class NativeDemo {

     public static void main(String []args){
        NativeClass n = new NativeClass();
        n.loadNativeLibrary();
     }
}
```

8. transient modifier

- transient is a modifier applicable only for variables.
- We can use transient keyword in serialization context.
- At the time of serialization, if we don't want to save the value of a particular variable due to security constraint then we should declare that variable as transient.
- At the time of serialization, JVM ignores original value of transient variables & save default value to the file; hence transient means not to serialize.

9. volatile modifier [Almost deprecated]

- volatile is a modifier applicable only for variables & we can't apply anywhere else.
- If the value of a variable keeps on changing by multiple threads, then there may be a chance of data inconsistency problem. We can solve this problem by using volatile modifier.
- If a variable is declared as volatile, then for every method, JVM will create a separate local copy. Every modification performed by a thread will take place in local copy so that there is no effect on the remaining threads.
- The main advantage of volatile keyword is we can overcome data inconsistency problem.
- The main disadvantage of volatile keyword is creating & maintaining a separate copy for every thread increases complexity of programming & creates performance problems; hence if there is no specific requirement, it is never recommended to use volatile keyword & it is almost deprecated.

Summary table for modifiers

	Classes						Interfaces		Enums	
Modifiers	**Outer**	**Inner**	**Constructor**	**Methods**	**Variables**	**Blocks**	**Outer**	**Inner**	**Outer**	**Inner**
public	Y	Y	Y	Y	Y	N	Y	Y	Y	Y
private	N	Y	Y	Y	Y	N	N	Y	N	Y
protected	N	Y	Y	Y	Y	N	N	Y	N	Y
default	Y	Y	Y	Y	Y	N	Y	Y	Y	Y
final	Y	Y	N	Y	Y	N	N	N	N	N
abstract	Y	Y	N	Y	N	N	Y	Y	N	N
static	N	Y	N	Y	Y	Y	N	Y	N	Y
synchronized	N	N	N	Y	N	Y	N	N	N	N
native	N	N	N	Y	N	N	N	N	N	N
strictfp	Y	Y	N	Y	N	N	Y	Y	Y	Y
transient	N	N	N	N	Y	N	N	N	N	N
volatile	N	N	N	N	Y	N	N	N	N	N

4. Interfaces

- Any service requirement specification (SRS) or any contract between client & service provider or 100% pure abstract class is nothing but interface.

Interface declaration & implementation

- Whenever we're implementing an interface, compulsory we should either provide implementation for each & every method of that interface or we should declare the class as abstract then next level child class is responsible to provide implementation.
- Every interface method is always public & abstract whether we're declaring or not, hence whenever we're implementing an interface method compulsory we should declare as public otherwise we will get compile time error.

extends Vs implements

- A class can extend only one class at a time.
- An interface can extend any number of interfaces simultaneously.
- A class can implement any number of interfaces simultaneously.
- A class can extend anther class & can implement any number of interfaces simultaneously.

Interface methods

- Every method present inside interface is always public & abstract whether we're declaring or not.

e.g.

```
interface Interf {

    void m1 ();  // public void m1(); or abstract void m1(); or public abstract void m1();
}
```

public – To make this method available to every implementation class.

abstract – Implementation class is responsible to provide implementation.

Interface variable

- An interface can contain variables, the main purpose of interface variable is to define requirement level constants.
- Every interface variable is always public, static & final whether we're declaring or not.

e.g.

```
interface Interf {

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

}
```

public – To make this variable available to every implementation class.

static – Without existing object also, implementation class can access this variable.

final – If one implementation class changes value then remaining classes will be effected. To restrict this, every interface variable is always final so that implementation class can't change value, but it can access.

- For interface variable, compulsory we should perform initialization at the time of declaration otherwise we will get compile time error.

Interface method & variable naming conflicts

1. Method naming conflicts

Case 1: If 2 interfaces contain a method with same signature & same return type then in the implementation class, we have to provide implementation for only one method.

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

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

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

Case 2: If 2 interfaces contain a method with same name but different argument types then in the implementation class, we have to provide implementation for both method & these methods acts as overloaded methods.

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

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

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

    public void m1 () {
        ….
    }
}
```

Overloaded methods

Case 3: If 2 interfaces contain a method with same signature but different return types then it is impossible to implement both interfaces simultaneously.

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

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

```
Not possible to implement both interfaces
simultaneously
```

- So yes, a Java class can implement any number of interfaces simultaneously but except for a particular case 3

2. Variable naming conflicts

- 2 interfaces can contain a variable with the same name & there may be a chance of variable naming conflicts but we can solve this problem by using interface names.

```
interface Left {
    int x = 777;
}
```

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

```
class Test implements Left, Right {
   System.out.println(Left.x);   // 777
   System.out.println(Right.x);  // 888
}
```

Marker interface

- If an interface doesn't contain any methods & by implementing that interface if our object will get some ability, such type of interfaces are called Marker interfaces.

  e.g.

  - **Serializable (I)** – By implementing Serializable interface, our objects can be saved to the file & can travel across the network,
  - **Cloneable (I)** – By implementing Cloneable interface, our objects are in a position to produce exactly duplicate cloned objects,
  - **RandomAccess (I)** etc.

- Internally JVM is responsible to provide required ability.
- We can create our own Marker interface but customization of JVM will be required.

Adapter classes

- Adapter class is a simple java class that implements an interface with only empty implementation.
- If we implement an interface, we should compulsorily provide implementation whether it is required or not for each & every method of that interface.
  - The problem with this approach is it increases the length of the code & reduces readability.
- We can solve this problem by using Adapter classes.

- Instead of implementing interface, if we extend adapter class, we have to provide implementation only for the required methods & we're not responsible for providing implementation for each & every method of the interface so the length of the code will be reduced.

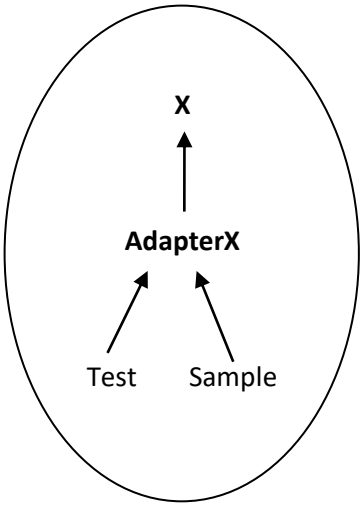
```
interface X {
     m1();
     m2();
     m3();
     m4();
     …….
     m100();
}
```

```
abstract class AdapterX implements X {
    m1() {}
    m2() {}
    m3() {}
    m4() {}
     …….
    m100() {}

}
```

```
class Test extends AdapterX {
    m3() {….}
}

class Sample extends AdapterX {
    m60() {….}
}
```

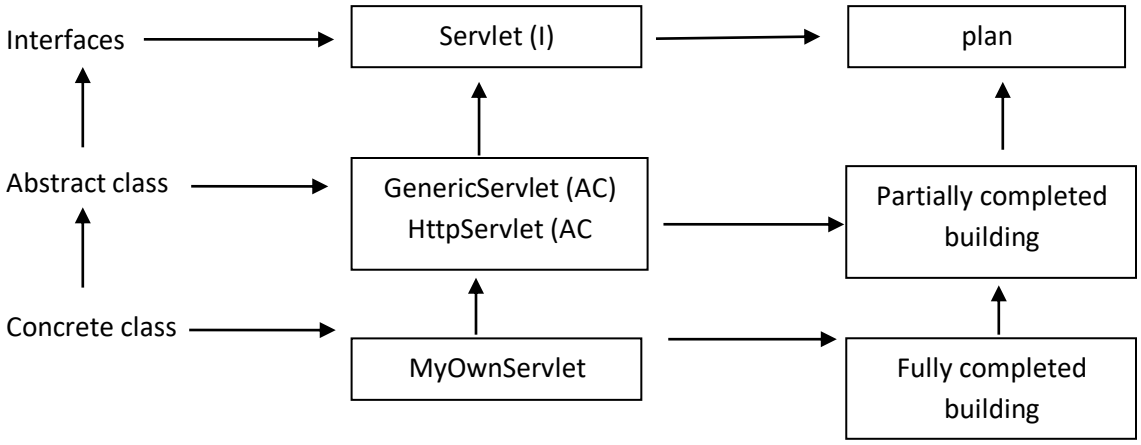


Note: Marker interface & Adapter classes simplifies complexity of programming & these are best utilities to the programmer & programmer's life will become simple.

Interface Vs Abstract class Vs Concrete class

- If we don't know anything about implementation, just we have requirement specification, we should go for interface.
  e.g. Servlet (I)
- If we are talking about implementation but not completely (partial implementation) then we should go for abstract class.
  e.g. GenericServlet (AC), HttpServlet (AC)

- If we are talking about implementation completely & ready to provide service then we should go for concrete class.
  e.g. MyOwnServlet



Interface Vs Abstract class

No.	Interface	Abstract class
1.	If we don't know anything about implementation & just we have requirement specification then we should go for interface.	If we're talking about implementation but not completely i.e. partial implementation then we should go for abstract class.
2.	Inside interface, every method is always public & abstract whether we're declaring or not; hence interface is 100% pure abstract class.	Every method present inside abstract class need not to be public & abstract; we can write concrete methods also.
3.	Every variable present inside interface is always public, static & final whether we're declaring or not.	Every variable present inside abstract class needs not to be public, static & final.
4.	For interface variables, compulsory we should perform initialization at the time of declaration only otherwise we will get compile time error.	For abstract class variables, we're not required to perform initialization at the time of declaration.
5.	Inside interface, we can't declare static & instance blocks.	Inside abstract class, we can declare static & instance blocks.
6.	Inside interface, we can't declare constructor.	Inside abstract class, we can declare constructor.

Q. Anyway we can't create object for abstract class but abstract class can contain constructor. What is the need?
Ans: - Abstract class constructor will be executed whenever we're creating child class object to perform initialization of child class object.

Approach 1: - Without having constructor in abstract class

```
abstract class Person {
    String name;
    int age;
    // ..... 100 properties

}
```

```
class Student extends Person {
    int rollno;
    Student(String name, int age, int
rollno) {
        this.name = name;
        this.age = age;
        //…. 100 properties

        this.rollno = rollno;

    }
}

Student s1 = new Student(101 properties)
```

```
Class Teacher extends Person {
    String subject;
    Teacher(String name, int age, String
subject) {
        this.name = name;
        this.age = age;
        //.... 100 properties

        this.subject = subject

   }
}

Teacher t = new Teacher(101 properties)
```

- More code & more redundancy. That's why this approach is not recommended.

Approach 2: - With constructor inside abstract class

```
abstract class Person {
    String name;
    int age;
    // ..... 100 properties

   Person (String name,  int age, ….) {
      this.name = name;
      this.age = age;
     // .. properties
    }
}
```

```
class Student extends Person {
    int rollno;
    Student(String name, int age, ….,int
rollno) {
        super(name, age,…. 100
properties)
        this.rollno = rollno;
    }
}

Student s1 = new Student(101 properties)
```

```
Class Teacher extends Person {
    String subject;
    Teacher(String name, int age,..,
String subject) {
        super(name, age,…. 100 properties)
        this.subject = subject
   }
}

Teacher t = new Teacher(101 properties)
```

- Less code & more code reusability. That's why this approach is recommended.
- Directly or indirectly, we can't create object for abstract class.

Q. Anyway we can't create objects for abstract class & interface but abstract class can contain constructor but interface doesn't contain. What's the reason?

Ans: - The main purpose of constructor is to perform initialization of instance variables.

- Abstract class can contain instance variables which are required for child object. To perform initialization of those instance variables constructor is required for abstract class.

- But every variable present inside interface is always public, static, final whether we're declaring or not & there is no chance of existing instance variable inside interface; hence constructor concept is not required for interface.

- Whenever we're creating child class object, parent object won't be created just parent class constructor will be executed for child object purpose only.

```
class P {
    P() {
        System.out.println(this.hashCode());
    }
}

class C extends P {
    C() {
        System.out.println(this.hashCode());
    }
}

public class Test {
    public static void main(String[] args) {
        C c = new C();
        System.out.println("Test class. C hashCode : " + c.hashCode());
    }
}
```

Output: (Same hashcode)

```
1829164700
1829164700
Test class. C hashCode : 1829164700
```

Q. Inside interface, every method is always abstract & we can take only abstract methods in abstract class also, then what is the difference between interface and abstract class? Is it possible to replace interface with abstract class?

Ans: - Yes, we can replace but if everything is abstract then it is highly recommended to go for interface not for abstract class.

Approach 1	Approach 2

`abstract class X {`
`}`
`class Test extends X {`
`}`
	`interface X {`
	`}`
	`class Test implements X {`
	`}`

While extending abstract class, it is not possible to extend any other class & hence we're missing inheritance benefit.	While implementing interface, we can extend some other class & hence we won't miss any inheritance benefit.
In this case, Object creation is costly. e.g. Test t = new Test(); - Whenever we're creating child class object, automatically parent constructor will execute & parent instance control flow will execute & so on.	In this case, object creation is not costly. e.g. Test t = new Test(); - It has nothing like parent constructor or parent instance control flow.