Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details,

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

- Abstract class
- Interface

Abstract class in Java

A class which is declared as **abstract** is known as an abstract class. It can have abstract and non-abstract methods.

A partially completed class is known as abstract class. When a class is not 100% completed we can make it an abstract class.

Points to Remember

- An abstract class must be declared with an **abstract** keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It can have constructor & static method also.
- It can have final methods which will force the subclass not to change the body of the method.

Example of abstract class

```
abstract class A
{
   // Abstract & non abstract method
}
```

Abstract Method in Java

A method which is declared as **abstract** and does not have implementation or doesn't have body is known as an abstract method.

Example

```
Abstract public void printStatus();          //no method body and abstract
```

If a method is defined without the body in Abstract class, child class is responsible to implement these methods.

Example of Abstract class that has an abstract method

```
abstract class Bike{
   abstract void run();
}
class Honda4 extends Bike{
       void run()
       {
         System.out.println("running safely");
       }
       public static void main(String args[]){
            Bike obj = new Honda4();
            obj.run();
      }
}
```

Abstract class having constructor, data member and methods

An abstract class can have a **data member**, **abstract method**, **method body (non-abstract method)**, **constructor**.

```
 abstract class Bike{
       Bike(){
         System.out.println("bike is created");
    }
    abstract void run();
    void changeGear(){
      System.out.println("gear changed");
    }
 }
 class Honda extends Bike{
```

```
    void run(){
        System.out.println("running safely..");
    }
}
class TestAbstraction2{
    public static void main(String args[]){
        Bike obj = new Honda();
        obj.run();
        obj.changeGear();
    }
}
```

Note: If there is an abstract method in a class, that class must be abstract.

Rule: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

Interface in Java

The interface in Java is a mechanism to achieve **abstraction.** An interface is a blue print of a class which has only abstract method with no body. It is also can't be instantiated like an abstract class.

Any service requirement specification, someone is responsible to provide the implementation.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Since **Java 8**, we can have **default** and **static methods** in an interface.

Since **Java 9**, we can have **private methods** in an interface.

Since java 8 & 9 version we can't say interface is a 100% Abstract class.

Why use Java interface?

There are mainly Two reasons to use interface. They are given below.

- It is used to achieve **abstraction**.
- By interface, we can support the functionality of **multiple inheritance.**

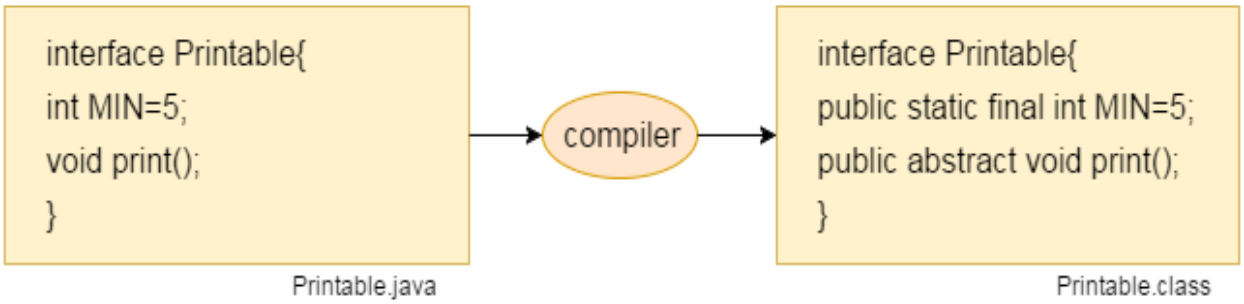
Declaration of Interface

```
interface <interface_name> {

    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

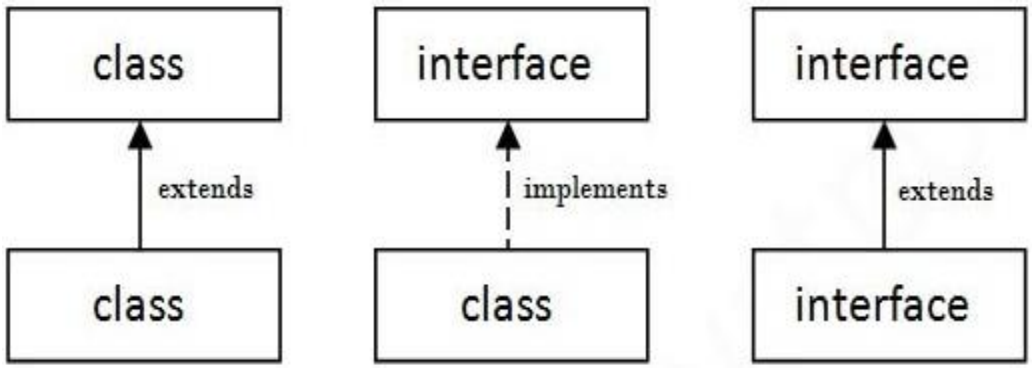
Note: ***The Java compiler adds public and abstract keywords before the interface method. and, it adds public, static and final keywords before data members.***

In other word, **Interface fields** are **public**, **static** and **final** by default, and the **methods** are **public** and **abstract**.



The relationship between classes and interfaces

As shown in the figure given below, a **class extends another class**, an **interface extends another interface**, but a **class implements** an **interface**.



```
interface printable{
   void print();
}
class A6 implements printable{
    public void print(){
       System.out.println("Hello");
     }
    public static void main(String args[]){
         A6 obj = new A6();
        obj.print();
    }
}
```

Java Interface Example: Drawable

In this example, the **Drawable** interface has only one method. Its implementation is provided by Rectangle and Circle classes.

```
interface Drawable{
   void draw();
}
class Rectangle implements Drawable{
      public void draw(){
        System.out.println("drawing rectangle");
      }
}
class Circle implements Drawable{
      public void draw(){
        System.out.println("drawing circle");
      }
}

class TestInterface1{
      public static void main(String args[]){
         Drawable d=new Circle();
         d.draw();
```

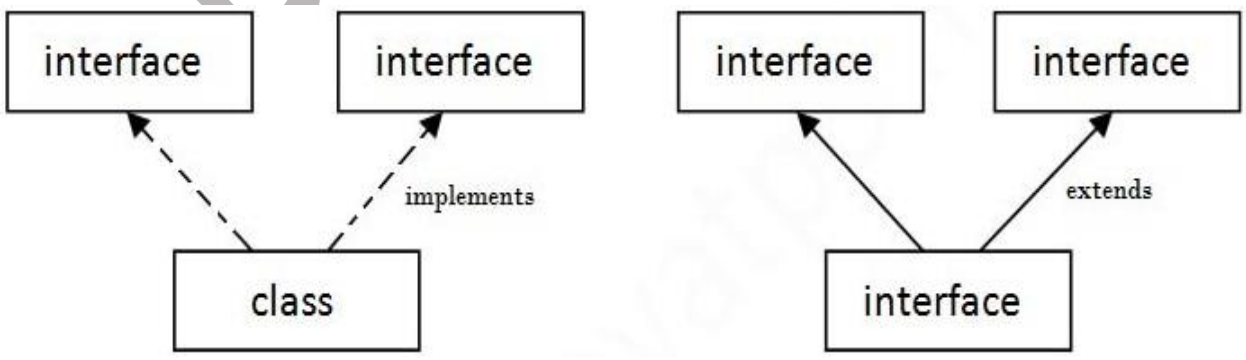
```
    }
}
```

Bank

```
interface Bank{
   float rateOfInterest();
}
class SBI implements Bank{
      public float rateOfInterest(){
          return 9.15f;
      }
}
class PNB implements Bank{
       public float rateOfInterest(){
           return 9.7f;
       }
}
class TestInterface2{
     public static void main(String[] args){
         Bank b=new SBI();
         System.out.println("ROI: "+b.rateOfInterest());
     }
}
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



```
interface Printable{
  void print();
}
interface Showable{
   void show();
}
class A7 implements Printable,Showable{
     public void print(){
         System.out.println("Hello");
     }
     public void show(){
         System.out.println("Welcome");
     }

     public static void main(String args[]){
         A7 obj = new A7();
         obj.print();
         obj.show();
    }
}
```

Interface inheritance

A class implements an interface, but one interface extends another interface.

```
interface Printable{
   void print();
}
interface Showable extends Printable{
   void show();
}
class TestInterface4 implements Showable{
     public void print(){
        System.out.println("Hello");
     }
```

```
    public void show(){
        System.out.println("Welcome");
    }

    public static void main(String args[]){
        TestInterface4 obj = new TestInterface4();
        obj.print();
        obj.show();
    }
}
```

Java 8 default method in java

```
interface Drawable{
    void draw();
   default void msg(){System.out.println("default method");}
}
class Rectangle implements Drawable{
    public void draw(){
      System.out.println("drawing rectangle");
    }
}
class TestInterfaceDefault{
    public static void main(String args[]){
       Drawable d=new Rectangle();
        d.draw();
         d.msg();
   }
}
```

Java 9 static method in java

```
interface Drawable{
  void draw();
  static int cube(int x){
    return x*x*x;
  }
```

```
}
class Rectangle implements Drawable{
    public void draw(){
        System.out.println("drawing rectangle");
    }
}
class TestInterfaceStatic{
      public static void main(String args[]){
          Drawable d=new Rectangle();
          d.draw();
          System.out.println(Drawable.cube(3));
       }
}
```

Q) What is marker or tagged interface?

An interface which has no member is known as a marker or tagged interface, for example, **Serializable**, **Cloneable**, **Remote**, etc.

```
public interface Serializable {
}
```

Difference Between Abstract class & Interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.

5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves **partial abstraction (0 to 100%)** whereas interface achieves **fully abstraction (100%).**

Package

A java package is a group of similar types of classes, interfaces and sub-packages.

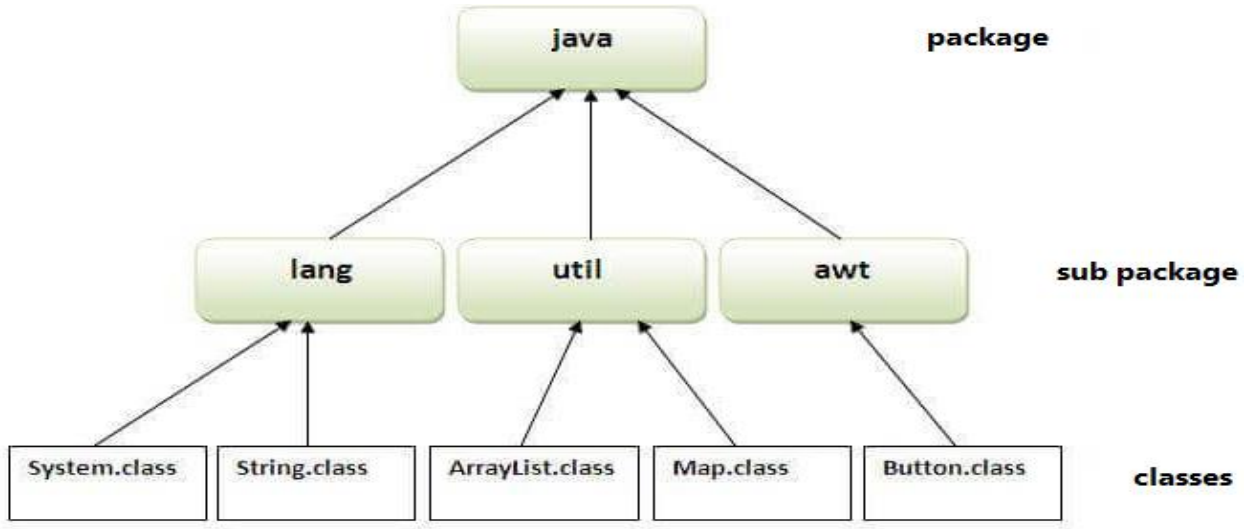
Package in java can be categorized in two form, **built-in package** and **user-defined** package.

There are many built-in packages such as **java**, **lang**, **awt**, **javax**, **swing**, **net**, **io**, **util**, **sql** etc.

There are almost 5000+ classes are available in java and each and every class must be inside some package. So, it is good programming practice to make our class in some package. It is standard form in java.

Advantage of Java Package

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.



How to create a package in java?

The **package** keyword is used to create a package in java.

Syntax

```
package package_name;
```

Eg:-

```
package com.bishal.patel;
```

NOTE: ***The Standard format of package name is in domain naming system in reverse order. However almost in every books any name is given.***

```
package com.bishal.patel;
public class SimplePackage{
 public static void main(String args[]){
   System.out.println("Welcome to package");
  }
}
```

How to compile java package?

```
javac -d directory javafilename
```

Example

```
javac -d . SimplePackage.java
```

-d *means destination*, **(dot).** *Means current working directory*. This statement creates a folder in the current working directory with package name.

com->Bishal->patel->SimplePackage.class

you can give any destination at the place of (dot). But usually we give only dot.

How to run java package program?

You need to use fully qualified name e.g. **java com.bishal.patel.SimplePackage** to run the class.

To compile: javac –d . SimplePackage.java

To run: java com.bishal.patel.SimplePackage

Two Conslusion be there in the package

```
package pack1;
package pack1;
public class SimplePackage{
    public static void main(String args[]){
        System.out.println("Welcome to package");
    }
}
```

How many .class file will create, where this .class file will be store. Is it valid or not ?

```
import java.util.Scanner;
package pack1;
public class SimplePackage{
    public static void main(String args[]){
      Scanner sc = new Scanner(System.in);
        System.out.println("Welcome to package");
    }
}
```

Is it valid, if yes then how if no then why?

Modifiers in Java

There are two types of modifiers in java.

- Access modifiers.
- Non-access modifiers.

In the we focus on access modifiers deeply.

Access Modifiers

The access modifiers in Java specifies the accessibility of a field, method, constructor, or class.

There are four types of Java access modifiers:

- **private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
- **default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
- **protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
- **public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many **non-access modifiers**, such as **static**, **abstract**, **synchronized**, **native**, **volatile**, **transient**, **final, strictfp** etc. Here, we are going to learn the access modifiers only.

Private

```
class A{
    private int data=40;
    private void msg(){
        System.out.println("Hello java");
    }
}
```

```
public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

Private constructor

```
class A{
   private A()      //private constructor
   {
    }
}
public class Simple{
 public static void main(String args[]){
   A obj=new A();     //Compile Time Error
 }
}
```

Default

```
package pack;
class A{
  void msg(){System.out.println("Hello");}
}
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();    //Compile Time Error
   obj.msg();      //Compile Time Error
  }
}
```

Protected

```
package pack;
public class A{
    protected void msg(){
        System.out.println("Hello");
    }
}
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Modifiers	Inner Class	Outer Class	Methods	Variable	Inner Interface	Outer Interface	Constructor
public	✓	✓	✓	✓	✓	✓	✓
Private	✗	✓	✓	✓	✗	✓	✓
protected	✗	✓	✓	✓	✗	✓	✓
Default	✓	✓	✓	✓	✓	✓	✓
Final	✓	✓	✓	✓	✗	✗	✗
abstract	✓	✓	✓	✗	✓	✓	✗
Static	✗	✓	✓	✓	✗	✓	✗

synchro	✗	✗	✓	✗	✗	✗	✗
native	✗	✗	✓	✗	✗	✗	✗
Strictfp	✓	✓	✓	✗	✓	✓	✗
transient	✗	✗	✗	✓	✗	✗	✗
Volatile	✗	✗	✗	✓	✗	✗	✗

Some Important Questions

- The modifiers which are applicable for inner classes but not for outer classes.
  - **private, protected, static**
- The modifiers which are applicable for classes but not for interface **final.**
- The modifiers which are applicable only for method and which can't use anywhere else **native.**
- The only modifiers which are applicable for constructor.
  - **public, protected, private, default.**
- The only applicable modifiers for local variable is **final.**

Class A
{
Class B
{
 	
}
}
	Class A
	{
	static interface B
	{
	
	}
	}

interface A
{
public static interface B
{
 	
}
}
	interface A
	{
	public static Class B
	{
	
	}
	}

By default, it should be whether we declare or not.

illegal Combinations of Modifiers

For method

Public ---> private, protected or vice versa

Abstract ---> final, static, private, native, synchronized, strictfp or vice versa

For variable

Public ---> private, protected or vice versa

Final ---> volatile or vice versa

For class

Public ---> private, protected or vice versa

Final ---> abstract or vice versa