# UNIT -6

# Abstract class and Interface

## Abstract class

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

**Abstract class**: is a restricted **class** that cannot be used to create objects (to access it, it must be inherited from another **class.**

## Abstraction in Java

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

It shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

### Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class
2. Interface

# Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

## *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 constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

---

# Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

**Example of abstract method**

```
1. abstract void printStatus();//no method body and abstract
```

## Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
1. abstract class Bike
2. {
3.   abstract void run();
4. }
5. class Honda4 extends Bike
6. {
7. void run()
8. {
9. System.out.println("running safely");
10.      }
11.      public static void main(String args[])
12.      {
13.       Bike obj = new Honda4();
14.       obj.run();
15.      }
16.      }
```

Output

```
running safely
```

## Another example of Abstract class

```
1. abstract class Shape
```

```
2. {
3. abstract void draw();
4. }
5. //In real scenario, implementation is provided by others i.e. unkn
   own by end user
6. class Rectangle extends Shape
7. {
8. void draw()
9. {
10.  System.out.println("drawing rectangle");
11.  }
12.  }
13.  class Circle1 extends Shape
14.  {
15.  void draw()
16.  {
17.  System.out.println("drawing circle");}
18.  }
19.  //In real scenario, method is called by programmer or user
20.  class TestAbstraction1
21.  {
22.  public static void main(String args[])
23.  {
24.  Shape s=new Circle1();//In a real scenario, object is provided
   through method, e.g., getShape() method
25.  s.draw();
26.  }
27.  }
```

Output

```
drawing circle
```

In this example, Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.

## Interface in Java

An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

You can say that interfaces can have abstract methods and variables. It cannot have a method body.

### Why use Java interface?

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

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

## How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public,

static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

```
1. interface <interface_name>
2. {
3.       Int x;
4.     Void somefunction();
5.
6.     // declare constant fields
7.     // declare methods that abstract
8.     // by default.
9. }
```

## Properties of interface

1. Interface must be declared with the key word **'interface'.**

2. All interface methods are implicitly **public** and **abstract**. In another words you don't need to actually type the public or abstract modifiers in the method declaration, but method is still always public and abstract
.

3. All variables defined in an interface are **public, static,** and **final.** In another words, interfaces can declare only constants, not instance variables.

4. Interface methods must not be **static**.

5. Because interface methods are abstract, they cannot be marked **final, strictfp,** or **native.**

6. An interfaces can extend one or more other interfaces.

7. An interface cannot implement another interface or class.

8. Interface types can be used polymorphically.

# Extending Interfaces

An interface contains variables and methods like a class but the methods in an interface are abstract by default unlike a class. An interface extends another interface like a class implements an interface in interface inheritance.

A program that demonstrates extending interfaces in Java is given as follows:

# Example

```
interface A {
   void funcA();
}
interface B extends A {
   void funcB();
}
class C implements B {
   public void funcA() {
      System.out.println("This is funcA");
   }
   public void funcB() {
      System.out.println("This is funcB");
```

```
    }
}
public class Demo {
    public static void main(String args[]) {
        C obj = new C();
        obj.funcA();
        obj.funcB();
    }
}
```

**Output**

```
This is funcA
This is funcB
```

Now let us understand the above program.

The interface A has an abstract method funcA(). The interface B extends the interface A and has an abstract method funcB(). The class C implements the interface B. A code snippet which demonstrates this is as follows:

# Implementing Interface in Java with Example

---

An interface is used as “superclass” whose properties are inherited by a class. A class can implement one or more than one interface by using a keyword implements followed by a list of interfaces separated by commas.

When a class implements an interface, it must provide an implementation of all methods declared in the interface and all its super interfaces.

An interface is used as “superclass” whose properties are inherited by a class. A class can implement one or more than one interface by using a keyword implements followed by a list of interfaces separated by commas.

When a class implements an interface, it must provide an implementation of all methods declared in the interface and all its super interfaces.

When implementation interfaces, there are several rules −

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

# Java Interface Example/Implementing Interfaces

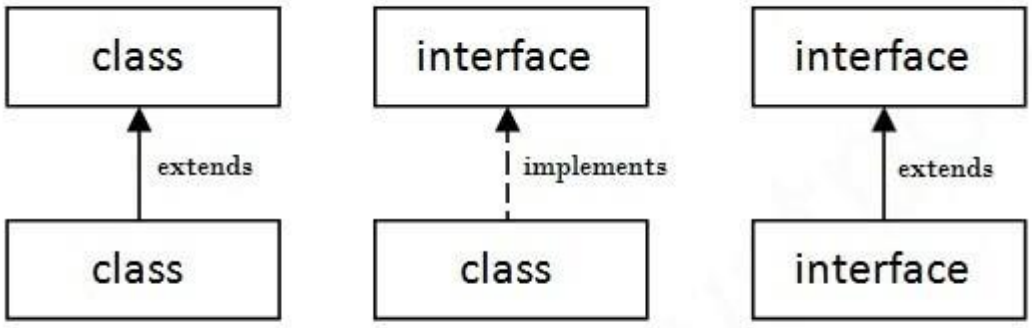
In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

```
1. interface printable
2. {
3. void print();  // method prototype
4. }
5. class A6 implements printable
   {
6. public void print()
7. {
8. System.out.println("Hello");
```

```
9. }
10.
11.     public static void main(String args[])
12.     {
13.     A6 obj = new A6();
14.     obj.print();
15.      }
16.     }
```

Output:

```
Hello
```




Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface.

| Abstract class | Interface |
| --- | --- |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstr** Since Java 8, it can have **defa** **static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple i** |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract** |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is use interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend anoth interface only. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemen keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface ar default. |

| | |
|---|---|
| 9)**Example:**<br>public abstract class Shape{<br>public abstract void draw();<br>} | **Example:**<br>public interface Drawable{<br>void draw();<br>} |

# Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract clas

```
1. //Creating interface that has 4 methods
2. interface A
3. {
4. void a();//by default, public and abstract
5. void b();
6. void c();
7. void d();
8. }
9.
10.  //Creating abstract class that provides the implementation of one meth
11.  abstract class B implements A
12.  {
13.  public void c()
14.  {
15.  System.out.println("I am C");
16.  }
17.  }
18.
19.  //Creating subclass of abstract class, now we need to provide the imple
```

```
20.  class M extends B
21.  {
22.  public void a()
23.  {
24.  System.out.println("I am a");
25.  }
26.  public void b()
27.  {
28.  System.out.println("I am b");
29.  }
30.  public void d()
31.  {
32.  System.out.println("I am d")
33.  ;}
34.  }
35.
36.  //Creating a test class that calls the methods of A interface
37.  class Test5
38.  {
39.  public static void main(String args[])
40.  {
41.  A a=new M();
42.  a.a();
43.  a.b();
44.  a.c();
45.  a.d();
46.  }
47.  }
```

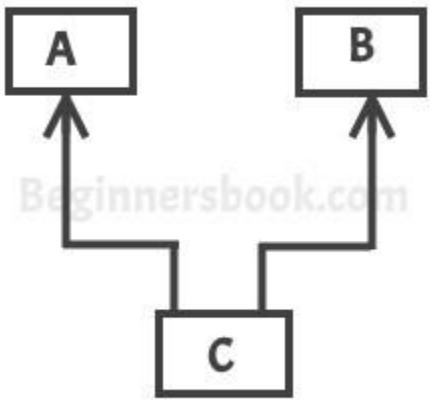
Output:

```
I am a
I am b
```

```
I am c
I am d
```

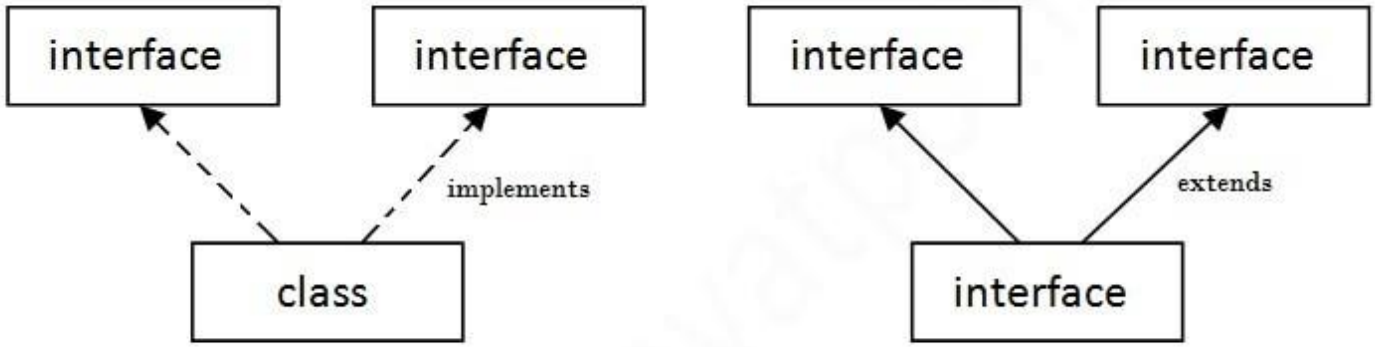
# Multiple inheritance in Java by interface

**Multiple Inheritance** is a feature of object oriented concept, where a class can **inherit** properties of more than one parent class. The problem occurs when there exist methods with same signature in both the super classes and subclass. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority.




Multiple Inheritance

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




**Multiple Inheritance in Java**

```
1. interface Printable
2. {
3. void print();
4. }
5. interface Showable
6. {
7. void show();
8. }
9. class A7 implements Printable,Showable
10.     {
11.     public void print()
12.     {
13.     System.out.println("Hello");
14.     }
15.     public void show()
16.     {
```

```
17.     System.out.println("Welcome");
18.     }
19.
20.     public static void main(String args[])
21.     {
22.     A7 obj = new A7();
23.     obj.print();
24.     obj.show();
25.      }
26.     }
```

```
Output:Hello
       Welcome
```

## Java packages

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.

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

## Built-in Packages

These packages consist of a large number of classes which are a part of Java **API**.Some of the commonly used built-in packages

are:
1) **java.lang:** Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.
2) **java.io:** Contains classed for supporting input / output operations.
3) **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
4) **java.applet:** Contains classes for creating Applets.
5) **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).
6) **java.net:** Contain classes for supporting networking operations.

# User-defined packages

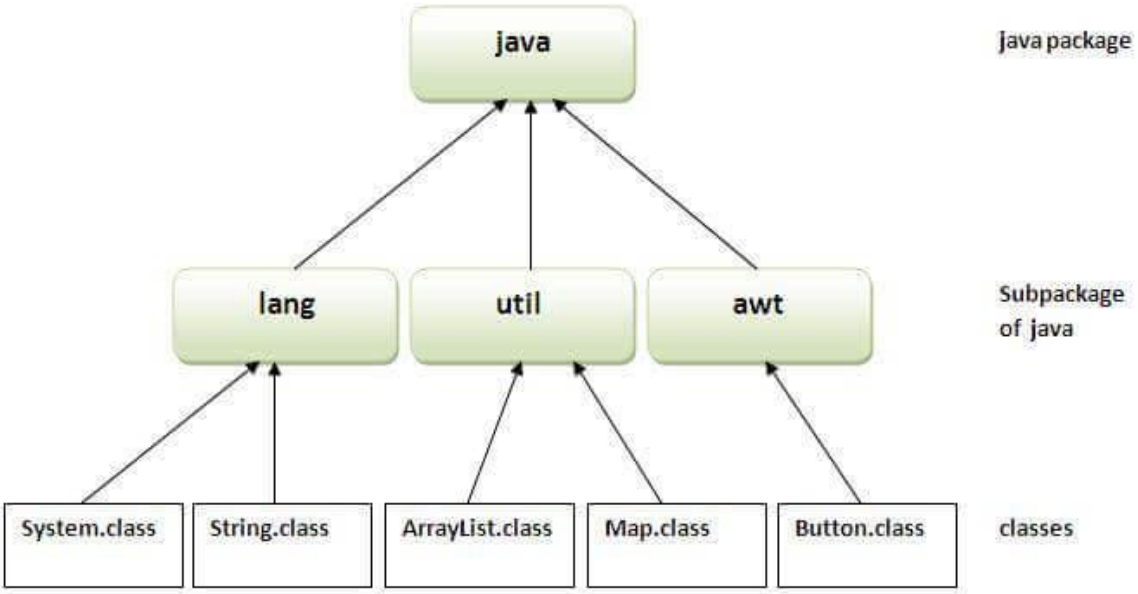
These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names**.

### Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.




# Naming conventions

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.

But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.

All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.

# Advantage of naming conventions in java

By using standard Java naming conventions, you make your code easier to read for yourself and other programmers. Readability of

Java program is very important. It indicates that less time is spent to figure out what the code does.

The following are the key rules that must be followed by every identifier:

- The name must not contain any white spaces.
- The name should not start with special characters like & (ampersand), $ (dollar), _ (underscore).

## Package

- It should be a lowercase letter such as java, lang.
- If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.

**Example :-**

```
1. package com.javatpoint; //package
2. class Employee
3. {
4. //code snippet
5. }
```

# Creating Packages

To create a package, follow the steps given below:

- Choose a package name according to the naming convention.
- Write the package name at the top of every source file (classes, interface, enumeration, and annotations).

- Remember that there must be only one package statement in each source file.

```
package mypack; //package declaration
class MyPackageClass //class definition
{
  public static void main(String[] args)
 {
    System.out.println("This is my package!");
  }
```

# To create a package, follow the steps given below:

1) Declare the package at the beginning of a file using the form package packagname;
2) Define the class that is to be put in the package and declare it public.
3) Create a subdirectory under the directory where the main source files are stored.
4) Store the listing as the classname.Javafile in the subdirectory created.
5) Compile the file.This creates.classfile in the subdirectory.

# Adding Classes to Packages

In order to put add Java classes to packages, you must do two things:

1. Put the Java source file inside a directory matching the Java package you want to put the class in.
2. Declare that class as part of the package.

Putting the Java source file inside a directory structure that matches the package structure, is pretty straightforward. Just create a source root directory, and inside that, create directories for each package and

subpackage recursively. Put the class files into the directory matching the package you want to add it to.

When you have put your Java source file into the correct directory (matching the package the class should belong to), you have to declare inside that class file, that it belongs to that Java package. Here is how you declare the package inside a Java source file:

```
package com.jenkov.navigation;

public class Page {
    ...
}
```

The first line in the code above (in bold) is what declares the class `Page` as belonging to the package `com.jenkov.navigation`.

# Static Import

> Static import is a feature introduced in **Java** programming language ( versions 5 and above ) that allows members ( fields and methods ) defined in a class as public **static** to be used in Java code without specifying the class in which the field is defined.
>
> The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

# Advantage of static import:

- Less coding is required if you have access any static member of a class oftenly.

### Disadvantage of static import:

- If you overuse the static import feature, it makes the program unreadable and unmaintainable.

## Simple Example of static import

```
1. import static java.lang.System.*;
2. class StaticImportExample{
3.   public static void main(String args[])
4. {
5.
6.    out.println("Hello");//Now no need of System.out
7.    out.println("Java");
8.
9.  }
10.     }
```

```
Output:Hello
       Java
```