# 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*](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/inheritance-in-java).

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

## 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:

**interface** <interface\_name>{

    // declare constant fields

    // declare methods that abstract

    // by default.

}

#### The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.

In other words, 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** Drawable{

**void** draw();

}

//Implementation: by second user

**class** Rectangle **implements** Drawable{

**public** **void** draw(){System.out.println("drawing rectangle");}

}

**class** Circle **implements** Drawable{

**public** **void** draw(){System.out.println("drawing circle");}

}

//Using interface: by third user

**class** TestInterface1{

**public** **static** **void** main(String args[]){

Drawable d=**new** Circle();//In real scenario, object is provided by method e.g. getDrawable()

d.draw();

}}

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();

 }

}

**TestInterface3.java**

**interface** Printable{

**void** print();

}

**interface** Showable{

**void** print();

}

**class** TestInterface3 **implements** Printable, Showable{

**public** **void** print(){System.out.println("Hello");}

**public** **static** **void** main(String args[]){

TestInterface3 obj = **new** TestInterface3();

obj.print();

 }

}

**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();

 }

}

**import** java.util.Scanner;

**interface** area

{

 **public** **void** dimensions();

 **public** **void** area();

}

**public** **class** interface1 **implements** area

{

 **int** length,breadth,area;

 **public** **void** dimensions()

 {

 Scanner s=**new** Scanner(System.in);

 System.out.print("Enter length:");

 length=s.nextInt();

 System.out.print("Enter breadth:");

 breadth=s.nextInt();

 }

 **public** **void** area()

 {

 area=length\*breadth;

 System.out.print("Area:"+area);

 }

 **public** **static** **void** main(String[] args)

 {

 Interface\_Implementation obj=**new** Interface\_Implementation();

 obj.dimensions();

 obj.area();

 }

}

# Java Program to Illustrates Use of Abstract Class and Method

**abstract** **class** Calculation

{

 **float** a = 12, b = 6, c;

 **abstract** **void** add();

 **void** subtract()

 {

 c = a - b;

 System.out.println("Result:"+c);

 }

 **abstract** **void** multiply();

 **void** divide()

 {

 c = a / b;

 System.out.println("Result:"+c);

 }

}

**public** **class** AbstractionDemo **extends** Calculation

{

 **void** add()

 {

 c = a + b;

 System.out.println("Result:"+c);

 }

 **void** multiply()

 {

 c = a \* b;

 System.out.println("Result:"+c);

 }

 **public** **static** **void** main(String[] args)

 {

 AbstractionDemo obj = **new** AbstractionDemo;

 obj.add();

 obj.subtract();

 obj.multiply();

 obj.divide();

 }

}

**Difference between class and interface in Java**

|  |  |
| --- | --- |
| Class | Interface |
| A class can be instantiated | An interface can never be instantiated |
| The class keyword is used to declare it | The **interface** keyword is used |
| The members of a class can be declared as private, public or protected | The members of an interface are always declared as public |
| Contains the concrete methods i.e methods with body | Contains abstract method i.e methods without the body |
| The extends keyword is used to inherit a class | The **implements**keyword is used to use an interface |
| Can contain [final](https://www.edureka.co/blog/final-finally-and-finalize-in-java/) and static methods | Cannot contain final or static methods |
| A Java class can have constructors | An interface cannot have constructors |
| A class can extend only one class but can implement any number of interfaces | An interface can extend any number of interfaces but cannot implement any interface |

# Difference between abstract class and 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. |
|  |  |
| 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();} |

# Java 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

Here, we will have the detailed learning of creating and using user-defined packages.

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.



Simple example of java package

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

//save as Simple.java

**package** mypack;

**public** **class** Simple{

 **public** **static** **void** main(String args[]){

    System.out.println("Welcome to package");

   }

}

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

|  |
| --- |
| **To Compile:** javac -d . Simple.java |
| **To Run:** java mypack.Simple |
| The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder. |

## How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.\*;
2. import package.classname;
3. fully qualified name.

#### 1) Using packagename.\*

If you use package.\* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

## Example of package that import the packagename.\*

//save by A.java

**package** pack;

**public** **class** A{

  **public** **void** msg(){System.out.println("Hello");}

}

//save by B.java

**package** mypack;

**import** pack.\*;

**class** B{

  **public** **static** **void** main(String args[]){

   A obj = **new** A();

   obj.msg();

  }

}

Output:Hello

#### 2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

## Example of package by import package.classname

1. //save by A.java

**package** pack;

**public** **class** A{

  **public** **void** msg(){System.out.println("Hello");}

}

//save by B.java

**package** mypack;

**import** pack.A;

**class** B{

  **public** **static** **void** main(String args[]){

   A obj = **new** A();

   obj.msg();

  }

}

Output:Hello

#### 3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

## Example of package by import fully qualified name

//save by A.java

**package** pack;

**public** **class** A{

  **public** **void** msg(){System.out.println("Hello");}

}

//save by B.java

**package** mypack;

**class** B{

  **public** **static** **void** main(String args[]){

   pack.A obj = **new** pack.A();//using fully qualified name

   obj.msg();

  }

}

Output:Hello

#### Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

## Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

#### The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

### Example of Subpackage

**package** com.javatpoint.core;

**class** Simple{

  **public** **static** **void** main(String args[]){

   System.out.println("Hello subpackage");

  }

}

|  |
| --- |
| **To Compile:** javac -d . Simple.java |
| **To Run:** java com.javatpoint.core.Simple |

Output:Hello subpackage