

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.

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

Java Interface also **represents the IS-A relationship**.

It cannot be instantiated just like the abstract class.

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

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

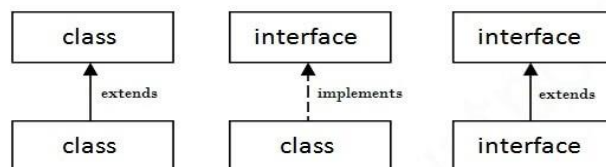
Why use Java interface?

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.

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**.



6.1 Defining Interfaces

In Java, defining an interface involves declaring a set of method signatures without providing their implementations. Here's an example:

```
java
// Defining an interface
public interface MyInterface {
    void method1();
    int method2(String input);
}
```

In this example, MyInterface declares two abstract methods (method1 and method2). Any class that implements this interface must provide concrete implementations for these methods.

6.2 Extending Interfaces

Java supports extending interfaces, allowing you to create a new interface that inherits the method declarations of one or more existing interfaces. Here's an example:

```
// Extending an interface
public interface MyExtendedInterface extends MyInterface {
    void additionalMethod();
}
```

In this case, `MyExtendedInterface` extends `MyInterface`, so it includes the methods from `MyInterface` and adds an additional abstract method `additionalMethod`. Classes implementing `MyExtendedInterface` must provide implementations for all three methods.

6.3 Implementing Interfaces

To use an interface in a class, the class needs to implement the interface by providing concrete implementations for all the methods declared in the interface. Here's an example:

```
// Implementing an interface
public class MyClass implements MyInterface {
    @Override
    public void method1() {
        // Implementation for method1
        System.out.println("Executing method1");
    }

    @Override
    public int method2(String input) {
        // Implementation for method2
        System.out.println("Executing method2 with input: " + input);
        return input.length();
    }
}
```

In this example, `MyClass` implements `MyInterface` and provides concrete implementations for both `method1` and `method2`.

6.4 Accessing Interface Variables

In Java, interface variables are implicitly public, static, and final, serving as constants accessible through the interface name. These variables act as shared constants that implementing classes can use.

Example:

```
// Interface with a variable
public interface MyInterfaceWithVariable {
    int MY_CONSTANT = 42;
}
```

You can access the interface variable like this:

```
public class MyClassUsingInterface implements MyInterfaceWithVariable {
    public void printConstant() {
        System.out.println("Interface constant value: " + MY_CONSTANT);
    }
}
```

```
}  
}
```

In the example of an interface named `MyInterfaceWithVariable` containing the constant `MY_CONSTANT`, implementing classes, like `MyClassUsingInterface`, can directly access and utilize this constant by referencing the interface name. This facilitates the creation of consistent values across multiple classes and reinforces the principle of encapsulation by centralizing shared constants within interfaces.

Java Interface Example

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

```
// Interface definition  
interface Printable {  
    void print();  
}  
  
// Class implementing the Printable interface  
class A6 implements Printable {  
    public void print() {  
        System.out.println("Hello");  
    }  
  
    public static void main(String args[]) {  
        // Creating an object of class A6  
        A6 obj = new A6();  
        // Calling the print method on the object  
        obj.print();  
    }  
}
```

Output:

Hello

Java Interface Example: Drawable

In this example, the `Drawable` interface has only one method. Its implementation is provided by `Rectangle` and `Circle` classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

File: `TestInterface1.java`

```
// Interface declaration: by first user  
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, the object is provided by a method, e.g., getDrawable()
        d.draw();
    }
}
```

Output:

drawing circle

Java Interface Example: Bank

Let's see another example of java interface which provides the implementation of Bank interface.

File: TestInterface2.java

```
// Interface declaration
interface Bank {
    float rateOfInterest();
}

// Implementation by SBI
class SBI implements Bank {
    public float rateOfInterest() {
        return 9.15f;
    }
}

// Implementation by PNB
class PNB implements Bank {
    public float rateOfInterest() {
        return 9.7f;
    }
}

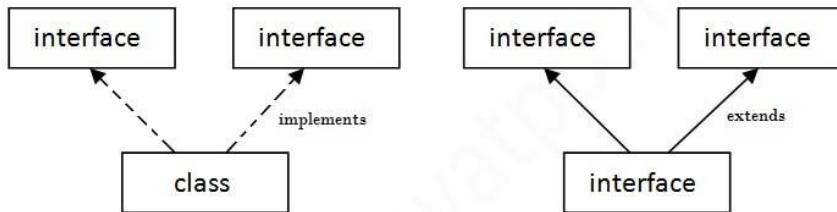
// Test class
class TestInterface2 {
    public static void main(String[] args) {
        Bank b = new SBI(); // Object created using interface
        System.out.println("ROI: " + b.rateOfInterest());
    }
}
```

Output:

ROI: 9.15

Multiple inheritance in Java by interface

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



Multiple Inheritance in Java

```
// Interface declaration
interface Printable {
    void print();
}

// Another interface declaration
interface Showable {
    void show();
}

// Class implementing multiple interfaces
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();
    }
}
```

Output:

Hello
Welcome

Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

```
// Interface declaration
interface Printable {
    void print();
}

// Another interface declaration
interface Showable {
    void print();
}

// Class implementing multiple interfaces with a common method
class TestInterface3 implements Printable, Showable {
    public void print() {
        System.out.println("Hello");
    }

    public static void main(String args[]) {
        TestInterface3 obj = new TestInterface3();
        obj.print();
    }
}
```

Output:
Hello

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class TestInterface1, so there is no ambiguity.

6.5 Introduction to 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. 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.

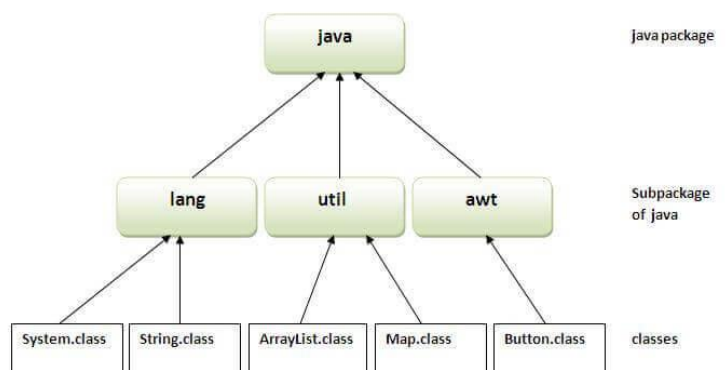
Advantage of Java Package

- 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");
    }
}
```



6.6 Creating a Package and naming convention

We use the **package** keyword as a very first line of code in the file for creating a package in Java4sz. We create a package by adding the package keyword as the very first line of code in a file. The syntax of creating a package in Java is as follows:

```
package com.javatpoint.packages
```

We highly recommend you to add only a unique type in a package. If we don't place the defined type in a package, they will be placed in the default or unnamed package.

These are the following disadvantages of putting types into unnamed packages:

- There are no benefits of having a package structure.
- The sub-packages are not possible.
- Importing the types in the default package from the other packages is not possible.
- There is no use of protected and package-private access scopes.

So, we should have to avoid the use of unnamed or default packages in RWA(Real World Applications). To learn more about packages, go through the following link:

Naming Conventions

For avoiding unwanted package names, we have some following naming conventions which we use in creating a package.

- The name should always be in the lower case.
- They should be period-delimited.
- The names should be based on the company or organization name.

In order to define a package name based on an organization, we'll first reverse the company URL. After that, we define it by the company and include division names and project names.

For example, if we want to create a package out of www.pec.com, we will reserve it in the following way:

```
com.pec
```

If we want to define a sub-package of the **com.pec**, we will do it in the following way:

```
com.pec.examples
```

How to use package members?

First, we define a class Test in a sub-package name examples:

```
// sub-package
package com.pec.packages.examples;
// create class Test in examples package
public class Test {
    private Long userId;
    private String password;

    // standard getters and setters
```

```
}
```

Now, if we want to use Test class from outside the package, we will import it in the following way:

```
import com.pec.packages.examples.Test;
```

In Java, we have some pre-defined classes and package available. We import the classes available in the pre-defined packages in the same way as we import classes in our own packages.

If we want to import ArrayList and List classes from the **util** package, we will import it in the following way:

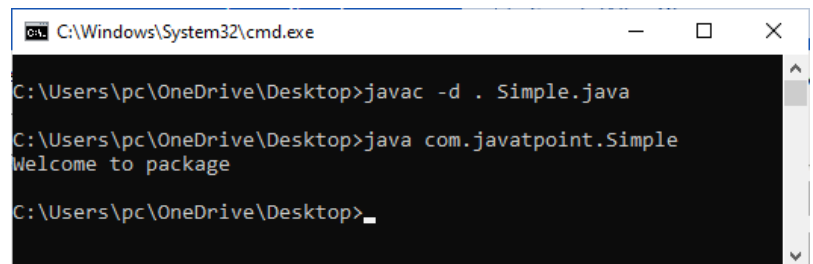
```
import java.util.ArrayList;
import java.util.List;
public class TestList {
    private List<String> names;
    public void addNames(String name) {
        if (names == null) {
            names = new ArrayList<String>();
        }
        names.add(name);
    }
}
```

Let's take an example to understand how we can create a package, run it and compile it using Java.

Simple.java

```
//save as Simple.java
package com.pec;
// create Simple class in mypackage
public class Simple{
    // main() method of Simple class
    public static void main(String args[]){
        // print package statement
        System.out.println("Welcome to package"
    );
    }
}
```

Output



```
C:\Windows\System32\cmd.exe
C:\Users\pc\OneDrive\Desktop>javac -d . Simple.java
C:\Users\pc\OneDrive\Desktop>java com.javatpoint.Simple
Welcome to package
C:\Users\pc\OneDrive\Desktop>_
```

6.6 Using Packages

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

```
//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.