

Packages and Interfaces: Defining a package, Finding packages and classpath, Importing packages, Defining an interface, Implementing interface, Extending interface.

Packages In Java

Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:

- Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding).

All we need to do is put related classes into packages. After that, we can simply write an import class from existing packages and use it in our program. A package is a container of a group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose. We can reuse existing classes from the packages as many time as we need it in our program.

How packages work?

Package names and directory structure are closely related. For example if a package name is *college.staff.cse*, then there are three directories, *college*, *staff* and *cse* such that *cse* is present in *staff* and *staff* is present inside *college*. Also, the directory *college* is accessible through CLASSPATH variable, i.e., path of parent directory of college is present in CLASSPATH. The idea is to make sure that classes are easy to locate. **Package naming conventions** : Packages are named in reverse order of domain names, i.e., org.geeksforgeeks.practice. For example, in a college, the recommended convention is college.tech.cse, college.tech.ee, college.art.history, etc.

0 seconds of 0 secondsVolume 0%

Adding a class to a Package : We can add more classes to a created package by using package name at the top of the program and saving it in the package directory. We need a new **java** file to define a public class, otherwise we can add the new class to an existing **.java** file and recompile it. **Subpackages**: Packages that are inside another package are the **subpackages**. These are not imported by default, they have to imported explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers. **Example** :

```
import java.util.*;
```

util is a subpackage created inside **java** package.

Accessing classes inside a package

Consider following two statements :

```
// import the Vector class from util package.
```

```
import java.util.vector;
```

```
// import all the classes from util package
```

```
import java.util.*;
```

- First Statement is used to import **Vector** class from **util** package which is contained inside **java**.
- Second statement imports all the classes from **util** package.

```
// All the classes and interfaces of this package
```

```
// will be accessible but not subpackages.
```

```
import package.*;
```

```
// Only mentioned class of this package will be accessible.
```

```
import package.classname;
```

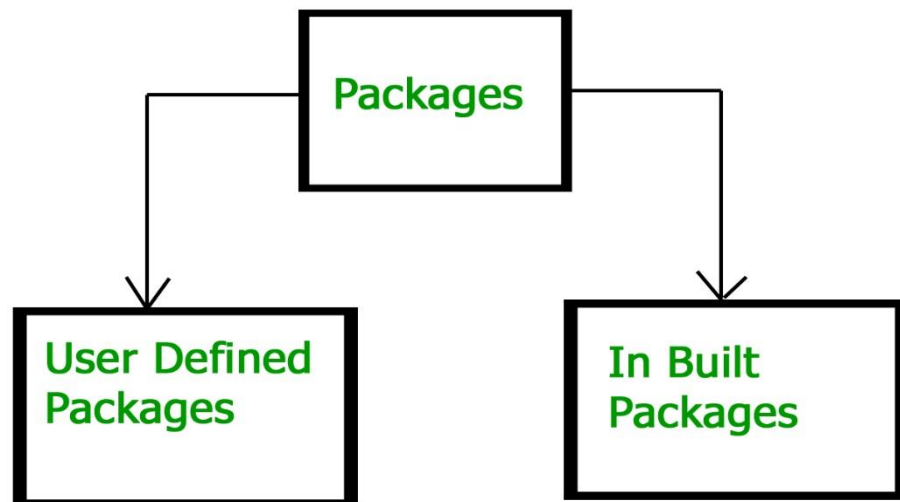
```
// Class name is generally used when two packages have the same
// class name. For example in below code both packages have
// date class so using a fully qualified name to avoid conflict
import java.util.Date;
import my.package.Date;
```

```
// Java program to demonstrate accessing of members when
// corresponding classes are imported and not imported.
import java.util.Vector;

public class ImportDemo
{
    public ImportDemo()
    {
        // java.util.Vector is imported, hence we are
        // able to access directly in our code.
        Vector newVector = new Vector();

        // java.util.ArrayList is not imported, hence
        // we were referring to it using the complete
        // package.
        java.util.ArrayList newList = new java.util.ArrayList();
    }

    public static void main(String arg[])
    {
        new ImportDemo();
    }
}
```



Types of 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. classes which define primitive data types, math operations). This package is automatically imported. 2) **java.io**: Contains classes 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**: Contains classes for implementing the components for graphical user interfaces (like button , ; menus etc). 6) **java.net**: Contains 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**.

// Name of the package must be same as the directory

// under which this file is saved

```
package myPackage;
```

```
public class MyClass
```

```
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
```

Now we can use the **MyClass** class in our program.

```
/* import 'MyClass' class from 'names' myPackage */
```

```
import myPackage.MyClass;
```

```
public class PrintName
```

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

```

{
    // Initializing the String variable
    // with a value
    String name = "GeeksforGeeks";

    // Creating an instance of class MyClass in
    // the package.
    MyClass obj = new MyClass();

    obj.getNames(name);
}
}

```

Note : **MyClass.java** must be saved inside the **myPackage** directory since it is a part of the package.

Using 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. Following program demonstrates **static import** :

```

// Note static keyword after import.
import static java.lang.System.*;

class StaticImportDemo
{
    public static void main(String args[])
    {
        // We don't need to use 'System.out'
        // as imported using static.
        out.println("GeeksforGeeks");
    }
}

```

Output:

GeeksforGeeks

Handling name conflicts

The only time we need to pay attention to packages is when we have a name conflict . For example both, java.util and java.sql packages have a class named Date. So if we import both packages in program as follows:

```

import java.util.*;

import java.sql.*;

```

//And then use Date class, then we will get a compile-time error :

Date today ; //ERROR-- java.util.Date or java.sql.Date?

The compiler will not be able to figure out which Date class do we want. This problem can be solved by using a specific import statement:

```
import java.util.Date;
```

```
import java.sql.*;
```

If we need both Date classes then, we need to use a full package name every time we declare a new object of that class. For Example:

```
java.util.Date deadLine = new java.util.Date();
```

```
java.sql.Date today = new java.sql.Date();
```

Directory structure

The package name is closely associated with the directory structure used to store the classes. The classes (and other entities) belonging to a specific package are stored together in the same directory. Furthermore, they are stored in a sub-directory structure specified by its package name. For example, the class Circle of package com.zzz.project1.subproject2 is stored as

“\$BASE_DIR\com\zzz\project1\subproject2\Circle.class”, where \$BASE_DIR denotes the base directory of the package. Clearly, the “dot” in the package name corresponds to a sub-directory of the file system. The base directory (\$BASE_DIR) could be located anywhere in the file system. Hence, the Java compiler and runtime must be informed about the location of the \$BASE_DIR so as to locate the classes. This is accomplished by an environment variable called CLASSPATH. CLASSPATH is similar to another environment variable PATH, which is used by the command shell to search for the executable programs. **Setting CLASSPATH:** CLASSPATH can be set by any of the following ways:

- CLASSPATH can be set permanently in the environment: In Windows, choose control panel ? System ? Advanced ? Environment Variables ? choose “System Variables” (for all the users) or “User Variables” (only the currently login user) ? choose “Edit” (if CLASSPATH already exists) or “New” ? Enter “CLASSPATH” as the variable name ? Enter the required directories and JAR files (separated by semicolons) as the value (e.g., “.;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar”). Take note that you need to include the current working directory (denoted by ‘.’) in the CLASSPATH. To check the current setting of the CLASSPATH, issue the following command:
 - > SET CLASSPATH
- CLASSPATH can be set temporarily for that particular CMD shell session by issuing the following command:
 - > SET CLASSPATH=.;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar
- Instead of using the CLASSPATH environment variable, you can also use the command-line option -classpath or -cp of the javac and java commands, for example,
 - > java -classpath c:\javaproject\classes com.abc.project1.subproject2.MyClass3

Illustration of user-defined packages:Creating our first package: File name – ClassOne.java

```
package package_name;

public class ClassOne {
    public void methodClassOne() {
        System.out.println("Hello there its ClassOne");
    }
}
```

Creating our second package: File name – ClassTwo.java

```
package package_one;
```

```
public class ClassTwo {
    public void methodClassTwo(){
        System.out.println("Hello there i am ClassTwo");
    }
}
```

Making use of both the created packages: File name – Testing.java

```
import package_one.ClassTwo;
import package_name.ClassOne;

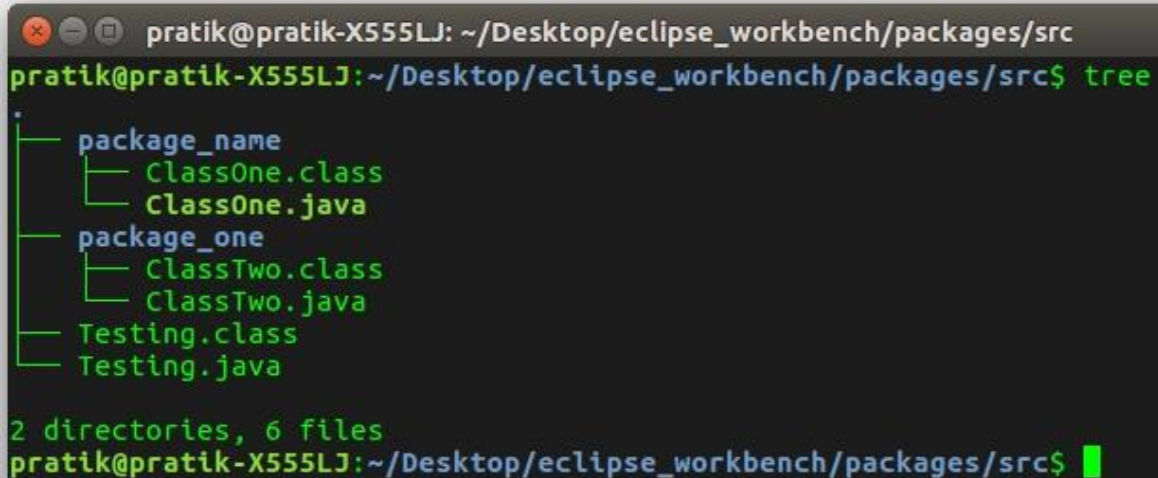
public class Testing {
    public static void main(String[] args){
        ClassTwo a = new ClassTwo();
        ClassOne b = new ClassOne();
        a.methodClassTwo();
        b.methodClassOne();
    }
}
```

Output:

Hello there i am ClassTwo

Hello there its ClassOne

Now having a look at the directory structure of both the packages and the testing class file:



A terminal window screenshot showing the directory structure of the project. The terminal title is 'pratik@pratik-X555LJ: ~/Desktop/eclipse_workbench/packages/src'. The command 'tree' has been executed, displaying the following structure:

```
pratik@pratik-X555LJ:~/Desktop/eclipse_workbench/packages/src$ tree
.
├── package_name
│   ├── ClassOne.class
│   └── ClassOne.java
├── package_one
│   ├── ClassTwo.class
│   └── ClassTwo.java
├── Testing.class
└── Testing.java

2 directories, 6 files
pratik@pratik-X555LJ:~/Desktop/eclipse_workbench/packages/src$
```

Important points:

1. Every class is part of some package.
2. If no package is specified, the classes in the file goes into a special unnamed package (the same unnamed package for all files).

3. All classes/interfaces in a file are part of the same package. Multiple files can specify the same package name.
4. If package name is specified, the file must be in a subdirectory called name (i.e., the directory name must match the package name).
5. We can access public classes in another (named) package using: **package-name.class-name**

Interfaces in Java

Read

Courses

Practice

Jobs

An **Interface in Java** programming language is defined as an abstract type used to specify the behavior of a class. An interface in Java is a blueprint of a behavior. A Java interface contains static constants and abstract methods.

What are Interfaces in Java?

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

When we decide on a type of entity by its behavior and not via attribute we should define it as an interface.

Syntax for Java Interfaces

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

To declare an interface, use the interface keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static, and final by default. A class that implements an interface must implement all the methods declared in the interface. To implement the interface, use the implements keyword.

Uses of Interfaces in Java

Uses of Interfaces in Java are mentioned below:

- *It is used to achieve total abstraction.*
- *Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.*
- *Any class can extend only 1 class, but can any class implement an infinite number of interfaces.*
- *It is also used to achieve loose coupling.*
- *Interfaces are used to implement abstraction.*

So, the question arises why use interfaces when we have abstract classes?

The reason is, abstract classes may contain non-final variables, whereas variables in the interface are final, public, and static.

// A simple interface


```
interface Player
{
    final int id = 10;
    int move();
}
```

Relationship Between Class and Interface

A class can extend another class similar to this an interface can extend another interface. But only a class can extend to another interface, and vice-versa is not allowed.

Difference Between Class and Interface

Although Class and Interface seem the same there have certain differences between Classes and Interface. The major differences between a class and an interface are mentioned below:

Class	Interface
In class, you can instantiate variables and create an object.	In an interface, you can't instantiate variables and create an object.
A class can contain concrete (with implementation) methods	The interface cannot contain concrete (with implementation) methods.
The access specifiers used with classes are private, protected, and public.	In Interface only one specifier is used- Public.

Implementation: To implement an interface, we use the keyword ***implements***

- Java

```
// Java program to demonstrate working of
// interface
```

```
import java.io.*;
```

```
// A simple interface
```

```
interface In1 {
```

```
    // public, static and final
```

```
    final int a = 10;
```

```
    // public and abstract
```

```
    void display();
```

```
}
```

```
// A class that implements the interface.
```

```
class TestClass implements In1 {
```

```
    // Implementing the capabilities of
```

```
    // interface.
```

```
    public void display(){
```

```
    System.out.println("Geek");
}

// Driver Code
public static void main(String[] args)
{
    TestClass t = new TestClass();
    t.display();
    System.out.println(a);
}
}
```

Output

Geek

10

Java Interfaces Examples

Let's consider the example of vehicles like bicycles, cars, bikes, etc they have common functionalities. So we make an interface and put all these common functionalities. And let's Bicycle, Bike, car, etc implement all these functionalities in their own class in their own way.

Below is the implementation of the above topic:

- Java

```
// Java program to demonstrate the
// real-world example of Interfaces
```

```
import java.io.*;
```

```
interface Vehicle {
```

```
    // all are the abstract methods.
```

```
    void changeGear(int a);
```

```
void speedUp(int a);  
void applyBrakes(int a);  
}
```

```
class Bicycle implements Vehicle{
```

```
    int speed;
```

```
    int gear;
```

```
    // to change gear
```

```
    @Override
```

```
    public void changeGear(int newGear){
```

```
        gear = newGear;
```

```
    }
```

```
    // to increase speed
```

```
    @Override
```

```
    public void speedUp(int increment){
```

```
        speed = speed + increment;
```

```
    }
```

```
    // to decrease speed
```

```
    @Override
```

```
    public void applyBrakes(int decrement){
```

```
        speed = speed - decrement;
```

```
    }
```

```
    public void printStates() {
```

```
        System.out.println("speed: " + speed  
        + " gear: " + gear);
```

```
}  
}
```

```
class Bike implements Vehicle {
```

```
    int speed;
```

```
    int gear;
```

```
    // to change gear
```

```
    @Override
```

```
    public void changeGear(int newGear){
```

```
        gear = newGear;
```

```
    }
```

```
    // to increase speed
```

```
    @Override
```

```
    public void speedUp(int increment){
```

```
        speed = speed + increment;
```

```
    }
```

```
    // to decrease speed
```

```
    @Override
```

```
    public void applyBrakes(int decrement){
```

```
        speed = speed - decrement;
```

```
    }
```

```
    public void printStates() {
```

```
        System.out.println("speed: " + speed
```

```
        + " gear: " + gear);
```

```
    }
```

```
}  
  
class GFG {  
  
    public static void main (String[] args) {  
  
        // creating an instance of Bicycle  
        // doing some operations  
        Bicycle bicycle = new Bicycle();  
        bicycle.changeGear(2);  
        bicycle.speedUp(3);  
        bicycle.applyBrakes(1);  
  
        System.out.println("Bicycle present state :");  
        bicycle.printStates();  
  
        // creating instance of the bike.  
        Bike bike = new Bike();  
        bike.changeGear(1);  
        bike.speedUp(4);  
        bike.applyBrakes(3);  
  
        System.out.println("Bike present state :");  
        bike.printStates();  
    }  
}
```

Output

Bicycle present state :

speed: 2 gear: 2

Bike present state :

speed: 1 gear: 1

Advantages of Interfaces in Java

The advantages of using interfaces in Java are as follows:

- Without bothering about the implementation part, we can achieve the security of the implementation.
- In Java, multiple inheritances are not allowed, however, you can use an interface to make use of it as you can implement more than one interface.

Multiple Inheritance in Java Using Interface

Multiple Inheritance is an OOPs concept that can't be implemented in Java using classes. But we can use multiple inheritances in Java using Interface. let us check this with an example.

Example:

- Java

```
// Java program to demonstrate How Diamond Problem
// Is Handled in case of Default Methods
```

```
// Interface 1
```

```
interface API {
    // Default method
    default void show()
    {

        // Print statement
        System.out.println("Default API");
    }
}
```

```
// Interface 2
```

```
// Extending the above interface
```

```
interface Interface1 extends API {
    // Abstract method
    void display();
}
```

```
// Interface 3
```

```
// Extending the above interface
```

```
interface Interface2 extends API {
    // Abstract method
    void print();
}
```

```
// Main class
```

```
// Implementation class code
```

```
class TestClass implements Interface1, Interface2 {
    // Overriding the abstract method from Interface1
```



```

public void display()
{
    System.out.println("Display from Interface1");
}
// Overriding the abstract method from Interface2
public void print()
{
    System.out.println("Print from Interface2");
}
// Main driver method
public static void main(String args[])
{
    // Creating object of this class
    // in main() method
    TestClass d = new TestClass();

    // Now calling the methods from both the interfaces
    d.show(); // Default method from API
    d.display(); // Overridden method from Interface1
    d.print(); // Overridden method from Interface2
}
}

```

Output

Default API

New Features Added in Interfaces in JDK 8

There are certain features added to Interfaces in JDK 8 update mentioned below:

1. *Prior to JDK 8, the interface could not define the implementation. We can now add default implementation for interface methods. This default implementation has a special use and does not affect the intention behind interfaces.*

Suppose we need to add a new function to an existing interface. Obviously, the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

Below is the implementation of the above point:

- Java

```
// Java program to show that interfaces can  
// have methods from JDK 1.8 onwards
```

```
interface In1
```

```
{  
    final int a = 10;  
    default void display()  
    {  
        System.out.println("hello");  
    }  
}
```

```
// A class that implements the interface.
```

```
class TestClass implements In1
```

```
{  
    // Driver Code  
    public static void main (String[] args)  
    {  
        TestClass t = new TestClass();  
        t.display();  
    }  
}
```

Output

hello

2. Another feature that was added in JDK 8 is that we can now define static methods in interfaces that can be called independently without an object.

Note: these methods are not inherited.

- Java

```
// Java Program to show that interfaces can
```

```
// have methods from JDK 1.8 onwards
```

```
interface In1
```

```
{  
    final int a = 10;  
    static void display()  
    {  
        System.out.println("hello");  
    }  
}
```

```
// A class that implements the interface.
```

```
class TestClass implements In1
```

```
{  
    // Driver Code  
    public static void main (String[] args)  
    {  
        In1.display();  
    }  
}
```

Output

hello

Extending Interfaces

One interface can inherit another by the use of keyword extends. When a class implements an interface that inherits another interface, it must provide an implementation for all methods required by the interface inheritance chain.

Program 1:

- Java

```
interface A {  
    void method1();  
    void method2();  
}  
  
// B now includes method1 and method2  
  
interface B extends A {  
    void method3();  
}  
  
// the class must implement all method of A and B.  
  
class gfg implements B {  
    public void method1()  
    {  
        System.out.println("Method 1");  
    }  
    public void method2()  
    {  
        System.out.println("Method 2");  
    }  
    public void method3()  
    {  
        System.out.println("Method 3");  
    }  
}
```

Program 2:

- Java

```
interface Student  
{  
    public void data();  
}  
  
class avi implements Student
```

```

{
    public void data ()
    {
        String name="avinash";
        int rollno=68;
        System.out.println(name);
        System.out.println(rollno);
    }
}

public class inter_face
{
    public static void main (String args [])
    {
        avi h= new avi();
        h.data();
    }
}

```

Output

avinash

68

In a Simple way, the interface contains multiple abstract methods, so write the implementation in implementation classes. If the implementation is unable to provide an implementation of all abstract methods, then declare the implementation class with an abstract modifier, and complete the remaining method implementation in the next created child classes. It is possible to declare multiple child classes but at final we have completed the implementation of all abstract methods.

In general, the development process is step by step:

Level 1 – interfaces: It contains the service details.

Level 2 – abstract classes: It contains partial implementation.

Level 3 – implementation classes: It contains all implementations.

Level 4 – Final Code / Main Method: It have access of all interfaces data.

Example:

- Java

```
// Java Program for
// implementation Level wise

import java.io.*;
import java.lang.*;
import java.util.*;

// Level 1
interface Bank {
    void deposit();
    void withdraw();
    void loan();
    void account();
}

// Level 2
abstract class Dev1 implements Bank {
    public void deposit()
    {
        System.out.println("Your deposit Amount :" + 100);
    }
}

abstract class Dev2 extends Dev1 {
    public void withdraw()
    {
        System.out.println("Your withdraw Amount :" + 50);
    }
}

// Level 3
class Dev3 extends Dev2 {
    public void loan() {}
    public void account() {}
}
```

```

}

// Level 4
class GFG {
    public static void main(String[] args)
    {
        Dev3 d = new Dev3();
        d.account();
        d.loan();
        d.deposit();
        d.withdraw();
    }
}

```

Output

Your deposit Amount :100

Your withdraw Amount :50

New Features Added in Interfaces in JDK 9

From Java 9 onwards, interfaces can contain the following also:

1. Static methods
2. Private methods
3. Private Static methods

Important Points in Java Interfaces

In the article, we learn certain important points about interfaces as mentioned below:

- We can't create an instance (interface can't be instantiated) of the interface but we can make the reference of it that refers to the Object of its implementing class.
- A class can implement more than one interface.
- An interface can extend to another interface or interface (more than one interface).
- A class that implements the interface must implement all the methods in the interface.
- All the methods are public and abstract. And all the fields are public, static, and final.
- It is used to achieve multiple inheritances.
- It is used to achieve loose coupling.
- Inside the Interface not possible to declare instance variables because by default variables are **public static final**.
- Inside the Interface, constructors are not allowed.
- Inside the interface main method is not allowed.
- Inside the interface, static, final, and private methods declaration are not possible.

Must Read

Frequently Asked Questions in Interfaces

1. What is a marker or tagged interface?

Tagged Interfaces are interfaces without any methods they serve as a marker without any capabilities.

2. How many Types of interfaces in Java?

Types of interfaces in Java are mentioned below:

- 1. Functional Interface*
- 2. Marker interface*