**Interfaces in Java**

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 class**. A Java interface contains 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 the 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**.

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

* Interfaces specify what a class must do and not how. It is the blueprint of the class.
* An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So, it specifies a set of methods that the class has to implement.
* If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
* A Java library example is Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

**Why do we use an Interface?**

* 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.
* 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();

}

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

// Java program to demonstrate working of

// interface

**import** java.io.\*;

// A simple interface

**interface** In1 {

    // public, static and final

**final** **int** a = 10;

    // always 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

**Real-World Example:**Let’s consider the example of vehicles like bicycle, car, bike………, they have common functionalities. So, we make an interface and put all these common functionalities. And lets Bicycle, Bike, car ….etc implement all these functionalities in their own class in their own way.

// 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:

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

New Features Added in Interfaces in JDK 8

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 in 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.

|  |
| --- |
| // 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

1. 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 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();

    }

}

**Important Points About Interface:**

* 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.

Implementation class is responsible to provide implementation. Also, We can’t use the following modifiers for interface methods.

* Private
* protected
* final
* static
* synchronized
* native
* strictfp

// A Simple Java program to demonstrate that interface methods must be public in

// implementing class

interface A

{

**void** fun();

}

**class** B implements A

{

    // If we change public to anything else,

    // we get compiler error

**public** **void** fun()

    {

        System.out.println("fun()");

    }

}

**class** C

{

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

    {

        B b = **new** B();

        b.fun();

    }

}

Output:

fun()

If we change fun() to anything other than public in class B, we get compiler error “attempting to assign weaker access privileges; was public”.

**Interface in a class**  
Interfaces (or classes) can have only public and default access specifiers when declared outside any other class (Refer [this](https://www.geeksforgeeks.org/g-fact-81/) for details). This interface declared in a class can either be default, public, protected not private. While implementing the interface, we mention the interface as**c\_name.i\_name** where **c\_name** is the name of the class in which it is nested and **i\_name** is the name of the interface itself.

// Java program to demonstrate working of

// interface inside a class.

**import** java.util.\*;

**class** Test

{

**interface** Yes

    {

**void** show();

    }

}

**class** Testing **implements** Test.Yes

{

**public** **void** show()

    {

        System.out.println("show method of interface");

    }

}

**class** A

{

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

    {

        Test.Yes obj;

        Testing t = **new** Testing();

        obj=t;

        obj.show();

    }

}

Interface can be private or protected or final.

The access specifier in above example is default. We can assign public, protected or private also. Below is an example of protected. In this particular example, if we change access specifier to private, we get compiler error because a derived class tries to access it.

// Java program to demonstrate protected

// specifier for nested interface.

**import** java.util.\*;

**class** Test

{

**protected** **interface** Yes

    {

**void** show();

    }

}

**class** Testing **implements** Test.Yes

{

**public** **void** show()

    {

        System.out.println("show method of interface");

    }

}

**class** A

{

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

    {

        Test.Yes obj;

        Testing t = **new** Testing();

        obj=t;

        obj.show();

    }

}

**Interface in another Interface**  
An interface can be declared inside another interface also. We mention the interface as **i\_name1.i\_name2** where **i\_name1** is the name of the interface in which it is nested and **i\_name2** is the name of the interface to be implemented.

// Java program to demonstrate working of interface inside another interface.

**import** java.util.\*;

**interface** Test

{

**interface** Yes

   {

**void** show();

   }

}

**class** Testing **implements** Test.Yes

{

**public** **void** show()

   {

      System.out.println("show method of interface");

   }

}

**class** A

{

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

   {

     Test.Yes obj;

     Testing t = **new** Testing();

     obj = t;

     obj.show();

   }

}

Interfaces Can Be Extended One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain.

Multiple Inheritance Issues

For example, assume that two interfaces called Alpha and Beta are implemented by a class called MyClass. What happens if both Alpha and Beta provide a method called reset( ) for which both declare a default implementation? Is the version by Alpha or the version by Beta used by MyClass?

To handle these and other similar types of situations, Java defines a set of rules that resolves such conflicts.

First, in all cases, a class implementation takes priority over an interface default implementation. Thus, if MyClass provides an override of the reset( ) default method, MyClass’ version is used. This is the case even if MyClass implements both Alpha and Beta. In this case, both defaults are overridden by MyClass’ implementation.

Second, in cases in which a class implements two interfaces that both have the same default method, but the class does not override that method, then an error will result.

Continuing with the example, if MyClass implements both Alpha and Beta, but does not override reset( ), then an error will occur.

With combination of extend and implement we can achieve Multiple Inheritance.