Interface in java

Interface

An interface is a collection of abstract methods and constants (i.e., static and final fields). It is used to achieve complete abstraction.

Note: Every interface in java is abstract by default. So, it is not compulsory to write abstract keyword with an interface.

Note: A class that implements an interface is called implementation class. A class can implement any number of interfaces in Java.

Syntax for interface

```
<Access specifiers> interface <interface_name> {
        // declare constant fields
        // declare methods that abstract
    }
```

Features of inheritance

1. It contains public abstract methods and public static final variables by default.
2. We must follow I to C design principle in java. It means every class must be implemented by some interfaces.
3. In company, Team Lead or Manager level people can design the interface then give it to developer for implementing it.
4. Before 1.7, interface does not have any method body.
5. 1.8 Declare the default & static method with body in interface.
6. 1.9 we can define the private methods in interface also.
7. We cannot create the object of interface.
8. In interface, we can just define the method only but implemented those methods into implemented class.
9. Java supports multiple inheritance in the terms of interfaces but not classes.
10. Interface does not have constructor.

Note:

a) Earlier to Java 8, an interface could not define any implementation whatsoever. An interface can only declare abstract methods.

b) Java 8 changed this rule. From Java 8 onwards, it is also possible to add a default implementation to an interface method.

Why do we use Interface?

1. In industry, architect-level people create interfaces, and then it is given to developers for writing classes by implementing interfaces provided.

Using interfaces is the best way to expose our project's API to some other projects. In other words, we can provide interface methods to the third-party vendors for their implementation.

For example, HDFC bank can expose methods or interfaces to various shopping carts.

Key points:

1. All methods of interfaces when implementing in a class must be declared as public otherwise you will get a compile-time error if any other modifier is specified.

2. Class extends class implements interface.

3. Class extends class implements Interface1, Interface2…

Example-

```
package com.abstra.interf;

 interface A {

     public abstract void demo();

     public abstract void example();

}
```

```
interface A{
   public abstract void demo ();  //allowed
   public void demo (); //allowed
   void demo (); //allowed
   abstract void demo (); //allowed
}
```

Note- if we don't write public or abstract in interface then JVM will insert it automatically.

}

```java
public class Z implements A {

	@Override
	public void demo() {

		System.out.println("this is demo method");

	}

	@Override
	public void example() {

		System.out.println("this is example method");

	}

}
package com.abstra.interf;

public class TestMain {

	public static void main(String[] args) {

		Z z = new Z();
		z.demo();
		z.example();
```

Relation between class and interface

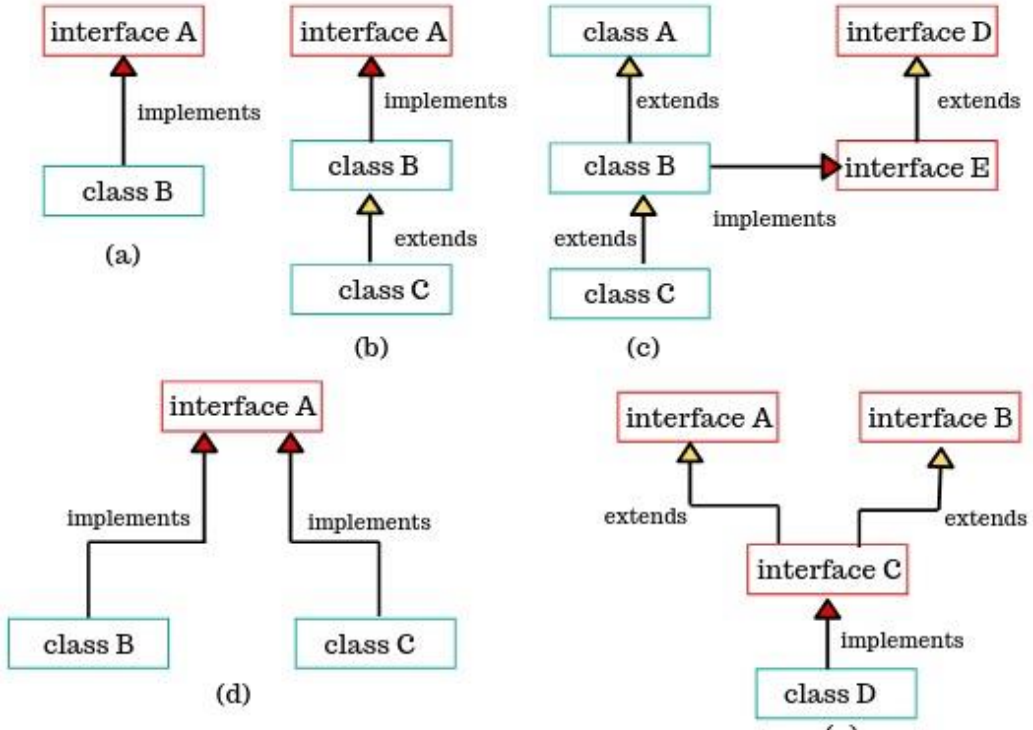


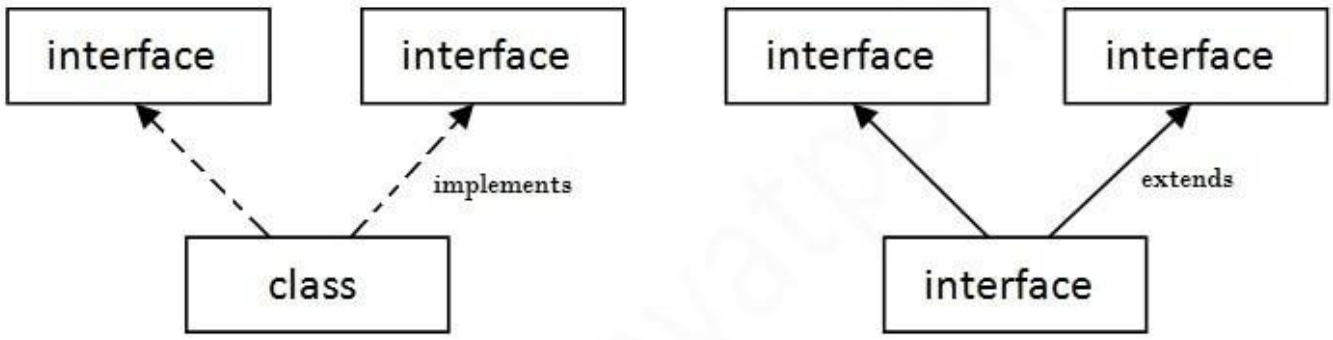
Fig: Various forms of interface implementation

Example 1

```
//Program for interface

interface Bank{

    float rateOfInterest();
}

class SBI  implements Bank{

public float rateOfInterest(){return 9.15f;}

}

class PNB implements Bank{

public float rateOfInterest(){return 9.7f;}

}

class TestInterface {

public static void main(String[] args){

Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());

   }
}
```

Multiple inheritance in Java by interface

When a class implements more than one interface, or an interface extends more than one interface, it is called multiple inheritance. Various forms of multiple inheritance are shown in the following figure.



Multiple Inheritance in Java

Example:

```
//Program for multiple inheritance in  interface

//1st interface
public interface Home
{
  void homeLoan();
 }
 //2nd interface
public interface Car
{
  void carLoan();
}
//3rd interface
public interface Education
{
  void educationLoan();
 }
```

```
// Multiple inheritance using multiple interfaces.
  public void homeLoan()
  {
    System.out.println("Rate of interest on home loan is 8.5%");
  }
public void carLoan()
{
   System.out.println("Rate of interest on car loan is 9.25%");
 }
public void educationLoan()
{
   System.out.println("Rate of interest on education loan is 10.45%");
 }

public static void main(String[] args)
{
   Loan l = new Loan();
    l.homeLoan();
    l.carLoan();
    l.educationLoan();
  }
}
```

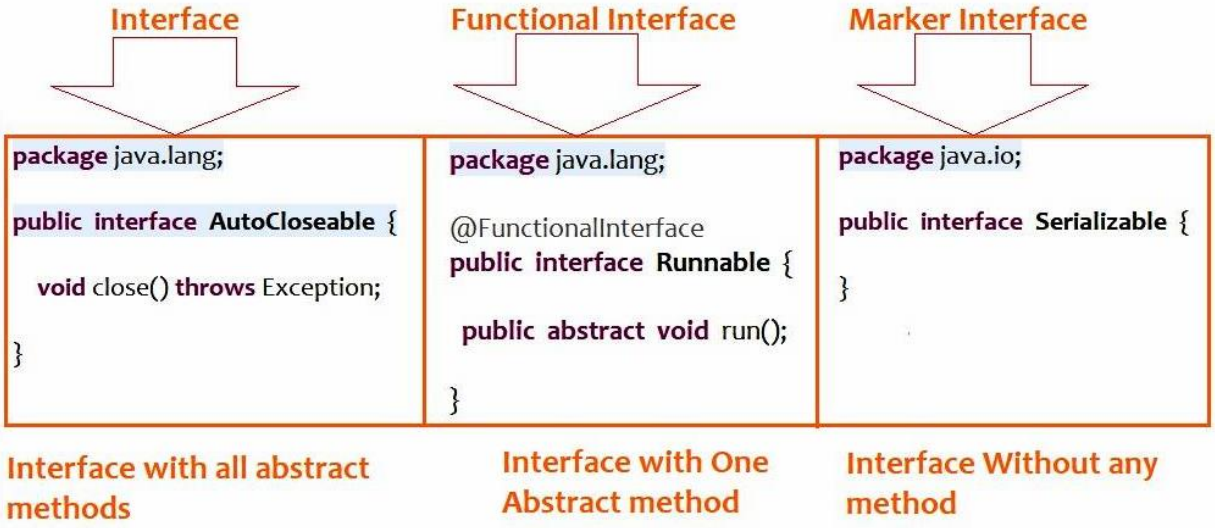
In Java, Multiple Inheritance is not supported through Class but it is possible by Interface, why?

As we have explained in the inheritance chapter, in multiple inheritance, subclasses are derived from multiple superclasses.

If two superclasses have the same method name then which method is inherited into subclass is the main confusion in multiple inheritance.

That's why Java does not support multiple inheritance in case of class. But, it is supported through an interface because there is no confusion. This is because its implementation is provided by the implementation class.

Types Of Interfaces In Java



Typically, we have three types of Interfaces till now.

1) Normal Interface

2) Marker Interface

3) Functional Interface

Normal Interface: Normal Interface is an interface which has either one or multiple number of abstract methods.

Marker interface: Marker Interface is an interface with no abstract method.

It is also known as a tagging interface and is used to indicate or inform the JVM that a class implementing this interface will have some special behavior.

uses, built-in (Serializable, Cloneable, and Remote Interfaces)

Functional Interface: Functional Interface is an interface which has only one abstract method. Further, it can have any number of static, default methods and even public methods of java.lang.Object class.

Runnable: contains only run() method

Comparable: contains only compareTo() method

ActionListener: contains only actionPerformed()

Callable: contains only call() method