Java - Interfaces

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways −

- An interface can contain any number of methods.

- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.

- The byte code of an interface appears in a **.class** file.

- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

- You cannot instantiate an interface.

- An interface does not contain any constructors.

- All of the methods in an interface are abstract.

- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.

- An interface is not extended by a class; it is implemented by a class.

- An interface can extend multiple interfaces.

Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface −

Example

Following is an example of an interface −

```
/* File name : NameOfInterface.java */

import java.lang.*;

// Any number of import statements


public interface NameOfInterface {

   // Any number of final, static fields

   // Any number of abstract method declarations\

}
```

Interfaces have the following properties −

- An interface is implicitly abstract. You do not need to use the **abstract**keyword while declaring an interface.

- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.

- Methods in an interface are implicitly public.

Example

```
/* File name : Animal.java */

interface Animal {

   public void eat();

   public void travel();

}
```

Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the

interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example

```java
/* File name : MammalInt.java */

public class MammalInt implements Animal {

   public void eat() {

      System.out.println("Mammal eats");

   }


   public void travel() {

      System.out.println("Mammal travels");

   }


   public int noOfLegs() {

      return 0;

   }


   public static void main(String args[]) {

      MammalInt m = new MammalInt();

      m.eat();

      m.travel();

   }

}
```

This will produce the following result −

Output

```
Mammal eats
```

When overriding methods defined in interfaces, there are several rules to be followed −

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.

- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.

- An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementation interfaces, there are several rules −

- A class can implement more than one interface at a time.

- A class can extend only one class, but implement many interfaces.

- An interface can extend another interface, in a similar way as a class can extend another class.

Extending Interfaces

An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

Example

```java
// Filename: Sports.java

public interface Sports {

   public void setHomeTeam(String name);

   public void setVisitingTeam(String name);

}


// Filename: Football.java

public interface Football extends Sports {
```

```java
   public void homeTeamScored(int points);

   public void visitingTeamScored(int points);

   public void endOfQuarter(int quarter);

}



// Filename: Hockey.java
public interface Hockey extends Sports {

   public void homeGoalScored();

   public void visitingGoalScored();

   public void endOfPeriod(int period);

   public void overtimePeriod(int ot);

}
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as −

Example

```java
public interface Hockey extends Sports, Event
```

Tagging Interfaces

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as −

Example

```
package java.util;

public interface EventListener

{}
```

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces −

Creates a common parent − As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

Adds a data type to a class − This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.