L36

Wednesday, February 2, 2022 12:06 AM

Understanding Interface, Using Interfaces

Understanding Interfaces

Q1. As the meaning of the word interface suggests, it is the point where two systems or subjects meet to interact.

Interface holds the same meaning in object oriented programming languages too. An interface defines a contract using which two classes/programs/systems can interact with each other.

In Java, interface is like a class, it is a reference type. It can have constants (they are not called fields), method signatures and nested members (we will learn more about nested members later).

The methods declared in interfaces should not contain the method body.
[Note: In **Java 8** and later versions, interfaces can contain default and static methods which can contain method body. We will learn more about them in later sections.]
Only the method signature should be present with a semicolon as terminator. For example:

```
interface Person {
	public static final int RETIREMENT_AGE = 60;  // example of a constant void setName(String name);  // only method signature without method bodyString getName();
	void setAge(int age);
	int getAge();
}
```

All methods declared in an interface are by default public and hence, we did not use the keyword public in the method signatures.

Interfaces cannot be instantiated. Meaning, Java compiler will throw out an error for a statement like : Person p = new Person();.

We can instantiate classes that implement the interface. Meaning, we can instantiate classes that provide the implementation for all the methods declared in the interface. For example:

```
public class Teacher implements Person {
	private String name;
	private int age;
	public void setName(String name) {
		this.name = name;
	}
	public String getName() {
		return name;
	}
	public void setAge(int age) {
		this.age = age;
	}
	public int getAge() {
		return age;
	}
}
```

Now, we can say Teacher t = new Teacher(); or
Person p = new Teacher();

Note: An interface can extend another interface, whereas a class implements one or more interfaces.

Select all the correct statements from below code:

```
interface Citizen {
	String getNationality();
}
interface Person {
	public static final int RETIREMENT_AGE = 60;
	void setName(String name);  // only method signature without method bodyString getName();
	void setAge(int age);
	int getAge();
}
public class Teacher implements Person, Citizen {  // statement 1private String name;
	private int age;
	private int nationality;
	public void setName(String name) {
		this.name = name;
	}
	public String getName() {
		return name;
	}
	public void setAge(int age) {
		this.age = age;
	}
	public int getAge() {
		return age;
	}
	public void setNationality(String nationality) {
		this.nationality = nationality;
	}
	public String getNationality() {
		return nationality;
	}
}
Citizen c = new Citizen();  // statement 2Citizen c = new Person();  // statement 3Citizen c = new Teacher();  // statement 4Person p = new Teacher();  // statement 5
```

- [ ] Statement 1 will result in compilation error, since class `Teacher` cannot implement two interfaces at the same time.
- [x] Statement 2 **will result** in a compilation error because we cannot instantiate an interface.
- [ ] Statement 3 **will not result** in a compilation error because we are trying to instantiate `Person` and assign to `Citizen`.
- [x] Statements 4 and 5 do not result in compilation errors.

Using Interfaces

Q1. An interface is primarily used to create a type.

In Java, a class is also a type which has behaviour attached to it.

However, in large object-oriented systems where multiple classes interact with each other an interface is the preferred means of creating a type. Classes are still needed which implement the type, however the publicly visible contract (methods) is published by the interface and the implementation is provided in the implementing classes.

By doing this we achieve what is called **loose-coupling**, where a class does not depend on the actual implementation (of another class) but rather depends on the contract (i.e. the methods) published through the interface. This ensures that the classes which rely on the interface, need not change whenever the underlying implementation in the implementing classes changes.

See and retype the below code to understand how the class InterfaceDemo need not know the difference in the implementation present in classes A and B, when it uses the interface.

Once you see the output you will realize that as long as the method in the interface Greeting is not changed, one need not change the code in class InterfaceDemo, even when the actual implementation code inside the method getGreetings in classes A and B change.

```
package q11283;
public class InterfaceDemo {
    public static void main(String[] args) {
        Greeting g1 = new A();
        Greeting g2 = new B();
        System.out.println(g1.getGreetings("Thor"));
        System.out.println(g2.getGreetings("Thor"));
    }
}
interface Greeting {
    String getGreetings(String name);
}
class A implements Greeting {
    public String getGreetings(String name) {
        return "Hi " + name;
    }
}
class B implements Greeting {
    public String getGreetings(String name) {
        return "Hola " + name;
    }
}

```

Q2. Write a Java program that implements an **interface**.

Create an interface called Car with two abstract methods String getName() and int getMaxSpeed(). Also declare one **default** method void applyBreak() which has the code snippet
System.out.println("Applying break on " + getName());
In the same interface include a **static** method Car getFastestCar(Car car1, Car car2), which returns **car1** if the **maxSpeed** of **car1** is greater than or equal to that of **car2**, else should return **car2**.

Create a class called BMW which implements the interface Car and provides the implementation for the abstract methods **getName()** and **getMaxSpeed()** (make sure to declare the appropriate fields to store **name** and **maxSpeed** and also the constructor to initialize them).

Similarly, create a class called Audi which implements the interface Car and provides the implementation for the abstract methods **getName()** and **getMaxSpeed()** (make sure to declare the appropriate fields to store **name** and **maxSpeed** and also the constructor to initialize them).

Create a **public** class called MainApp with the **main()** method.
Take the input from the command line arguments. Create objects for the classes BMW and Audi then print the fastest car.

Note:
Java 8 introduced a new feature called default methods or defender methods, which allow developers to add new methods to the interfaces without breaking the existing implementation of these interface. These **default** methods can also be overridden in the implementing classes or made abstract in the extending interfaces. If they are not overridden, their implementation will be shared by all the implementing classes or sub interfaces.

Below is the syntax for declaring a default method in an **interface** :

```
public default void methodName() {
        System.out.println("This is a default method in interface");
}
```

Similarly, **Java 8** also introduced static methods inside interfaces, which act as regular static methods in classes. These allow developers group the utility functions along with the interfaces instead of defining them in a separate helper class.

Below is the syntax for declaring a static method in an **interface** :

```
public static void methodName() {
        System.out.println("This is a static method in interface");
}
```

```java
package q11284;
interface Car {
    String getName();
    int getMaxSpeed();
    default void applyBreak() {
        System.out.println("Applying break on " + getName());
    }
    static void getFastestCar(Car car1, Car car2) {
        if (car1.getMaxSpeed() >= car2.getMaxSpeed()) {
            System.out.println("Fastest car is : " + car1.getName());
        }
        else {
            System.out.println("Fastest car is : " + car2.getName());
        }
    }
}
class BMW implements Car {
    String name;
    int maxSpeed;
    public BMW(String name, int maxSpeed) {
        this.name = name;
        this.maxSpeed = maxSpeed;
    }
    public String getName() {
        return name;
    }
    public int getMaxSpeed() {
        return maxSpeed;
    }
}
class Audi implements Car {
    String name;
    int maxSpeed;
    public Audi(String name, int maxSpeed) {
        this.name = name;
        this.maxSpeed = maxSpeed;
    }
    public String getName() {
        return name;
    }
    public int getMaxSpeed() {
        return maxSpeed;
    }
}
public class MainApp {
    public static void main(String args[]) {
        int value = Integer.parseInt(args[1]);
        int value1 = Integer.parseInt(args[3]);

        BMW bmw = new BMW(args[0], value);

        Audi audi = new Audi(args[2], value1);

        Car.getFastestCar(bmw, audi);
    }
}
```