Java Interface

In this tutorial, we will learn about Java interfaces. We will learn how to implement interfaces and when to use them in detail with the help of examples.

An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).

We use the `interface` keyword to create an interface in Java. For example,

```
interface Language {
  public void getType();

  public void getVersion();
}
```

Here,

- `Language` is an interface.
- It includes abstract methods: `getType()` and `getVersion()`.

Implementing an Interface

Like abstract classes, we cannot create objects of interfaces.

To use an interface, other classes must implement it. We use the `implements` keyword to implement an interface.

Example 1: Java Interface

```java
interface Polygon {
  void getArea(int length, int breadth);
}

// implement the Polygon interface
class Rectangle implements Polygon {

  // implementation of abstract method
  public void getArea(int length, int breadth) {
    System.out.println("The area of the rectangle is " + (length * breadth));
  }
}

class Main {
  public static void main(String[] args) {
    Rectangle r1 = new Rectangle();
    r1.getArea(5, 6);
  }
}
```

Output

```
The area of the rectangle is 30
```

In the above example, we have created an interface named `Polygon`. The interface contains an abstract method `getArea()`.

Here, the `Rectangle` class implements `Polygon`. And, provides the implementation of the `getArea()` method.

Example 2: Java Interface

```java
// create an interface
interface Language {
  void getName(String name);
}

// class implements interface
class ProgrammingLanguage implements Language {

  // implementation of abstract method
  public void getName(String name) {
    System.out.println("Programming Language: " + name);
  }
}

class Main {
  public static void main(String[] args) {
    ProgrammingLanguage language = new ProgrammingLanguage();
    language.getName("Java");
  }
}
```

Output

```
Programming Language: Java
```

In the above example, we have created an interface named `Language`. The interface includes an abstract method `getName()`.

Here, the `ProgrammingLanguage` class implements the interface and provides the implementation for the method.

Implementing Multiple Interfaces

In Java, a class can also implement multiple interfaces. For example,

```
interface A {
  // members of A
}

interface B {
  // members of B
}

class C implements A, B {
  // abstract members of A
  // abstract members of B
}
```

Extending an Interface

Similar to classes, interfaces can extend other interfaces. The `extends` keyword is used for extending interfaces. For example,

```
interface Line {
  // members of Line interface
}

// extending interface
interface Polygon extends Line {
  // members of Polygon interface
  // members of Line interface
}
```

Here, the `Polygon` interface extends the `Line` interface. Now, if any class implements `Polygon`, it should provide implementations for all the abstract methods of both `Line` and `Polygon`.

Extending Multiple Interfaces

An interface can extend multiple interfaces. For example,

```
interface A {
   ...
}
interface B {
   ...
}

interface C extends A, B {
   ...
}
```

Advantages of Interface in Java

Now that we know what interfaces are, let's learn about why interfaces are used in Java.

- Similar to abstract classes, interfaces help us to achieve **abstraction in Java**.

  Here, we know `getArea()` calculates the area of polygons but the way area is calculated is different for different polygons. Hence, the implementation of `getArea()` is independent of one another.
- Interfaces **provide specifications** that a class (which implements it) must follow.

  In our previous example, we have used `getArea()` as a specification inside the interface `Polygon`. This is like setting a rule that we should be able to get the area of every polygon.

  Now any class that implements the `Polygon` interface must provide an implementation for the `getArea()` method.
- Interfaces are also used to achieve multiple inheritance in Java. For example,

```
interface Line {
…
}

interface Polygon {
…
}

class Rectangle implements Line, Polygon {
…
}
```

Here, the class `Rectangle` is implementing two different interfaces. This is how we achieve multiple inheritance in Java.

> **Note**: All the methods inside an interface are implicitly `public` and all fields are implicitly `public static final`. For example,
>
> ```
> interface Language {
>
>   // by default public static final
>   String type = "programming language";
>
>   // by default public
>   void getName();
> }
> ```

---

default methods in Java Interfaces

With the release of Java 8, we can now add methods with implementation inside an interface. These methods are called default methods.

To declare default methods inside interfaces, we use the `default` keyword. For example,

```
public default void getSides() {
   // body of getSides()
}
```

Why default methods?

Let's take a scenario to understand why default methods are introduced in Java.

Suppose, we need to add a new method in an interface.

We can add the method in our interface easily without implementation. However, that's not the end of the story. All our classes that implement that interface must provide an implementation for the method.

If a large number of classes were implementing this interface, we need to track all these classes and make changes to them. This is not only tedious but error-prone as well.

To resolve this, Java introduced default methods. Default methods are inherited like ordinary methods.

Let's take an example to have a better understanding of default methods.

Example: Default Method in Java Interface

```
interface Polygon {
  void getArea();

  // default method
  default void getSides() {
    System.out.println("I can get sides of a polygon.");
  }
}

// implements the interface
class Rectangle implements Polygon {
  public void getArea() {
    int length = 6;
    int breadth = 5;
    int area = length * breadth;
    System.out.println("The area of the rectangle is " + area);
  }

  // overrides the getSides()
  public void getSides() {
    System.out.println("I have 4 sides.");
  }
}

// implements the interface
class Square implements Polygon {
  public void getArea() {
    int length = 5;
    int area = length * length;
    System.out.println("The area of the square is " + area);
  }
}

class Main {
  public static void main(String[] args) {

    // create an object of Rectangle
    Rectangle r1 = new Rectangle();
    r1.getArea();
    r1.getSides();

    // create an object of Square
    Square s1 = new Square();
    s1.getArea();
    s1.getSides();
  }
}
```

Output

```
The area of the rectangle is 30
I have 4 sides.
The area of the square is 25
I can get sides of a polygon.
```

In the above example, we have created an interface named `Polygon`. It has a default method `getSides()` and an abstract method `getArea()`.

Here, we have created two classes `Rectangle` and `Square` that implement `Polygon`.

The `Rectangle` class provides the implementation of the `getArea()` method and overrides the `getSides()` method. However, the `Square` class only provides the implementation of the `getArea()` method.

Now, while calling the `getSides()` method using the `Rectangle` object, the overridden method is called. However, in the case of the `Square` object, the default method is called.

---

private and static Methods in Interface

The Java 8 also added another feature to include static methods inside an interface.

Similar to a class, we can access static methods of an interface using its references. For example,

```
// create an interface
interface Polygon {
  staticMethod(){..}
}

// access static method
Polygon.staticMethod();
```

> **Note**: With the release of Java 9, private methods are also supported in interfaces.
>
> We cannot create objects of an interface. Hence, private methods are used as helper methods that provide support to other methods in interfaces.

Practical Example of Interface

Let's see a more practical example of Java Interface.

```java
// To use the sqrt function
import java.lang.Math;

interface  Polygon {
   void getArea();

 // calculate the perimeter of a Polygon
   default void getPerimeter(int... sides) {
      int perimeter = 0;
      for (int side: sides) {
         perimeter += side;
      }

   System.out.println("Perimeter: " + perimeter);
   }
}

class Triangle implements Polygon {
   private int a, b, c;
   private double s, area;

// initializing sides of a triangle
   Triangle(int a, int b, int c) {
      this.a = a;
      this.b = b;
      this.c = c;
      s = 0;
   }

// calculate the area of a triangle
   public void getArea() {
      s = (double) (a + b + c)/2;
      area = Math.sqrt(s*(s-a)*(s-b)*(s-c));
      System.out.println("Area: " + area);
   }
}

class Main {
   public static void main(String[] args) {
      Triangle t1 = new Triangle(2, 3, 4);

// calls the method of the Triangle class
      t1.getArea();

// calls the method of Polygon
      t1.getPerimeter(2, 3, 4);
   }
}
```

Output

```
Area: 2.9047375096555625
Perimeter: 9
```

In the above program, we have created an interface named `Polygon`. It includes a default method `getPerimeter()` and an abstract method `getArea()`.

We can calculate the perimeter of all polygons in the same manner so we implemented the body of `getPerimeter()` in `Polygon`.

Now, all polygons that implement `Polygon` can use `getPerimeter()` to calculate perimeter.

However, the rule for calculating the area is different for different polygons. Hence, `getArea()` is included without implementation.

Any class that implements `Polygon` must provide an implementation of `getArea()`.