

Interface Enhancements In Java 8 – Java Functional Interface

We explored all about **Interfaces in Java** in our last tutorial. We have introduced and covered the basic concepts of interfaces in Java including the multiple interfaces.

Before Java 8, interfaces were permitted to have only abstract methods and static and final variables. The abstract methods are by default public and need to be overridden by the class that implements an interface.

So interface was mainly a contract and was only involved with constants (static & final) and abstract methods.

What You Will Learn: [\[hide\]](#)

Interfaces Changes In Java 8

- Static Method In Interface In Java

- Interface Default Method

- Default Methods And Multiple Inheritance

Java 8 Functional Interfaces

- #1) Predicate

- #2) BinaryOperator

- #3) Function

Class Vs Interface In Java

Java Extends Vs Implements

- Can Abstract Class Implement Interface In Java

- When To Use Abstract Class And Interface In Java

 - When to use an Interface:

 - When to use an Abstract Class:

Interface Vs Abstract Class In Java

- Enum Inheritance In Java

- Frequently Asked Questions

Conclusion

- Recommended Reading

Interfaces Changes In Java 8

The Java 8 release introduces or allows us to have static and default methods in the interfaces. Using default methods in an interface, the developers can add more methods to the interfaces. This way they do not disturb or change the classes that implement the interface.

Java 8 also allows the interface to have a static method. Static methods are the same as those we define in classes. Note that the static method cannot be overridden by the class that implements the interface.

The introduction of static and default methods in the interface made it easier to alter the interfaces without any problems and also made interfaces easier to implement.

Java 8 also introduces “Lambda Expressions” inside functional interfaces. Besides, from Java 8 onwards there are more built-in functional interfaces added in Java.

In this tutorial, we will discuss all these additions to the interfaces in Java 8 and will also discuss some of the differences between various Java concepts like abstract classes, extends keyword, etc. with the interfaces.

Static Method In Interface In Java

Interfaces can also have methods that can have definitions. These are the static methods in the interface. The static methods are defined inside the interface and they cannot be overridden or changed by the classes that implement this interface.

We can call these static methods by directly using the interface name.

The following example demonstrates the use of the static method.

```
//interface declaration
interface TestInterface {
    // static method definition
    static void static_print() {
        System.out.println("TestInterface::static_print (");
    }
    // abstract method declaration
    void nonStaticMethod(String str);
}
```

```
}

// Interface implementation
class TestClass implements TestInterface {
    // Override interface method
    @Override
    public void nonStaticMethod(String str) {
        System.out.println(str);
    }
}

public class Main{
    public static void main(String[] args) {
        TestClass classDemo = new TestClass();

        // Call static method from interface
        TestInterface.static_print();

        // Call overridden method using class object
        classDemo.nonStaticMethod("TestClass::nonStaticMethod ()");
    }
}
```

Output:

```
TestInterface::static_print ()
TestClass::nonStaticMethod ()
```

The above program has a TestInterface. It has a static method named 'static_print' and also a non-static method named the nonstaticmethod.

We have implemented the TestInterface in TestClass and overridden nonStaticMethod. Then in the main method, we call static_print method directly using the TestInterface and nonStaticMethod using the object of the TestClass.

Interface Default Method

As already mentioned, interfaces before Java 8 permitted only abstract methods. Then we would provide this method implementation in a separate class. If we had to add a new method to the interface, then we have to provide its implementation code in the same class.

Hence if we altered the interface by adding a method to it, the implementation class also would change.

This limitation was overcome by Java 8 version that allowed the interfaces to have default methods. The default methods in a way provide backward compatibility to the existing interfaces and we need not alter the implementation class. The default methods are also known as “virtual extension method” or “defender methods”.

Default methods are declared by using the keyword “default” in the declaration. The declaration is followed by the definition of the method. We can override the default method as it is available to the class that implements the interface.

In the same way, we can invoke it using the implementation class object from the interface directly without overriding it.

```
interface TestInterface {
    // abstract method
    public void cubeNumber(int num);

    // default method
    default void print()
    {
        System.out.println("TestInterface :: Default method");
    }
}

class TestClass implements TestInterface {
    // override cubeNumber method
    public void cubeNumber(int num)
    {
        System.out.println("Cube of given number " + num+ ":" + num*num*num);
    }
}

class Main{
    public static void main(String args[])
    {
        TestClass obj = new TestClass();
        obj.cubeNumber(5);

        // call default method print using class object
        obj.print();
    }
}
```

Output:

```
Cube of given number 5:125
TestInterface :: Default method
```

The above Java program demonstrates the default method in the interface. In the main method, note that we can call the default method of the interface using the class object. This is because as the class implements the interface, the default method is available for the class as well.

Note: We could have overridden the print () method also in the implementation class. Note that if overridden, then the access modifier of the default method will change to public in the implementation class.

Default Methods And Multiple Inheritance

There might arise a situation in case of multiple interfaces wherein each interface might have a default method with the same prototype. In such a case, the compiler does not know which method to invoke.

When this situation arises in which the default method has the same prototype in all the interfaces, then the solution is to override the method in implementation class so that when the implementation class object calls the default method, the compiler invokes the method implemented in the class.

The following Java program demonstrates the use of the default method with multiple interfaces.

```
//Interface_One
interface Interface_One{
    //defaultMethod
    default void defaultMethod(){
        System.out.println("Interface_One::defaultMethod");
    }
}

//Interface_Two
interface Interface_Two{
    //defaultMethod
    default void defaultMethod(){
        System.out.println("Interface_Two::defaultMethod");
    }
}

class TestExample implements Interface_One, Interface_Two{
```

```
public void disp(String str){
    System.out.println("String is: "+str);
}
//override defaultMethod to take care of the ambiguity
public void defaultMethod(){
    System.out.println("TestExample::defaultMethod");
}
}
class Main{
    public static void main(String[] args) {
        TestExample obj = new TestExample();

        //call the default method
        obj.defaultMethod();
    }
}
```

Output:

```
TestExample::defaultMethod
```

In the above program, we have overridden the default method (which has the same prototype in both the interfaces) in the implementation class. This way when we call the default method from the main method using the object of the implementation class, the overridden method is invoked.

Java 8 Functional Interfaces

A functional interface is an interface that has only one abstract method. It can contain any number of default and static methods but the abstract method it contains is exactly one. Additionally, a functional interface can have declarations of object class methods.

Functional Interface is known as “**Single Abstract Method Interface**” or “**SAM Interface**”. SAM interface is a new feature in Java.

In a Java program, the presence of a functional interface is indicated by using a **@FunctionalInterface** annotation. When the compiler encounters this annotation, then it knows that the interface that is following this annotation is functional. Hence if it

contains more than one abstract method, then it flashes an error.

The annotation **@FunctionalInterface** however, is not mandatory in Java.

The following program demonstrates the Functional Interface in Java:

```
//declare a functional interface
@FunctionalInterface //annotation indicates it's a functional interface
interface function_Interface{
    void disp_msg(String msg); // abstract method
    // Object class methods.
    int hashCode();
    String toString();
    boolean equals(Object obj);
}
//implementation of Functional Interface
class FunctionalInterfaceExample implements function_Interface{
    public void disp_msg(String msg){
        System.out.println(msg);
    }
}
class Main{
    public static void main(String[] args) {
        //create object of implementation class and call method
        FunctionalInterfaceExample finte = new FunctionalInterfaceExample();
        finte.disp_msg("Hello, World!!!");
    }
}
```

Output:

```
Hello, World!!!
```

The functional interface in the above program has a single abstract method and also has an object class method declaration like hashCode, toString, and equals. In the class that implements this interface, the abstract method is overridden. In the main method, we create an object of implementation class and use the method.

Interfaces like Runnable and Comparable are examples of functional interfaces provided in Java. Java 8 allows us to assign lambda expressions to the functional interface object.

The following example program demonstrates this.

```
class Main{
    public static void main(String args[])
    {
        // use lambda expression to create the object
        new Thread(()->
            {System.out.println("New thread created with functional interface");}).start();
    }
}
```

Output:

```
New thread created with functional interface
```

Java 8 also provides many built-in functional interfaces in java.util.function package.

These built-in interfaces are described below:

#1) Predicate

This is a functional interface in Java that has a single abstract method test. The ‘test’ method returns the boolean value after testing the specified argument.

Given below is the prototype for the test method of the Predicate interface.

```
public interface Predicate {
    public boolean test(T t);
}
```

#2) BinaryOperator

BinaryOperator interface provides an abstract method ‘apply’ which accepts two arguments and returns a resultant value of the same type as the arguments.

The prototype for the accept method is:

```
public interface BinaryOperator {  
    public T apply (T x, T y);  
}
```

#3) Function

The Function interface is a functional interface that also has an abstract method named 'apply'. This apply method, however, takes a single argument of type T and returns a value of type R.

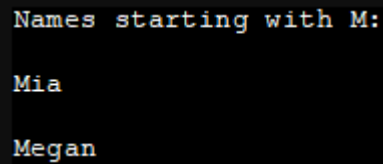
The prototype of the apply method is as follows:

```
public interface Function {  
    public R apply(T t);  
}
```

The following Java program demonstrates the above built-in Functional Interface Predicate.

```
import java.util.*;  
import java.util.function.Predicate;  
  
class Main  
{  
    public static void main(String args[])  
    {  
        // create a list of strings  
        List<String> names =  
            Arrays.asList("Karen", "Mia", "Sydney", "Lacey", "Megan");  
  
        // declare string type predicate and use lambda expression to create object  
        Predicate<String> p = (s)->s.startsWith("M");  
  
        System.out.println("Names starting with M:");  
        // Iterate through the list  
    }  
}
```

```
for (String st:names)
{
    // test each entry with predicate
    if (p.test(st))
        System.out.println(st);
}
}
```

Output:A screenshot of a terminal window with a black background and white text. The text shows the output of a Java program: "Names starting with M:", followed by "Mia" and "Megan" on separate lines.

```
Names starting with M:
Mia
Megan
```

As we can see in the above program we have a list of strings. Using the functional interface Predicate, we test if the item in the string starts with M and if it does, then it prints the name.

Class Vs Interface In Java

Although class and interface are similar as they have similar syntax, these two entities have more differences than similarities.

Let's list down some of the differences between class and interface in Java.

Class	Interface
Keyword 'class' is used to create a class.	The interface is created using the keyword 'interface'.
Classes do not support multiple inheritance in Java.	Interfaces support multiple inheritance in Java.
The class contains the constructors.	Interfaces do not contain constructors.
The class cannot contain abstract methods.	Interfaces contain only abstract methods.

Class	Interface
The class can have variables and methods that are default, public, private, or protected.	The interface has only public variables and methods by default.
It is not mandatory to associate non-access modifiers with variables of the class.	Interfaces can have variables that are either static or final.
We can instantiate and create objects from a class.	An interface cannot be instantiated.
We can inherit another class from a class.	We cannot inherit a class from the interface.
The class can be inherited using the keyword 'extends'.	The interface can be implemented by another class using the 'implements' keyword. It can be inherited by another interface using 'extends' keyword.

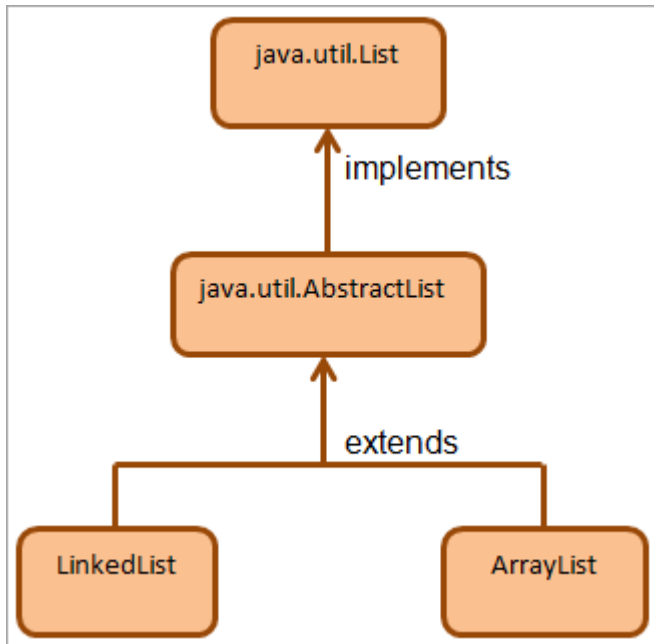
Java Extends Vs Implements

'extends'	'implements'
A class uses 'extends' keyword to inherit from another class.	'implements' keyword is used by a class to implement an interface.
A class inheriting another class may or may not override all the methods of the parent class.	The class implementing the interface must override all the abstract methods of the interface.
We can extend only one class at a time using the extends keyword.	We can implement multiple interfaces using the 'implements' keyword.
An interface can extend another interface using 'extends' keyword.	An interface cannot implement another interface using 'implements' keywords.

Can Abstract Class Implement Interface In Java

Yes, an abstract class can implement an interface using the 'implements' keyword. The abstract class need not implement all interface abstract methods. But overall it's a good design practice to have an interface with all abstract methods, then an abstract class implementing this interface, and then the concrete classes.

Given below is an example of such an implementation in Java.



Here `java.util.List` is an interface. This interface is implemented by `java.util.AbstractList`. Then this `AbstractList` class is extended by two concrete classes i.e. `LinkedList` and `ArrayList`.

If `LinkedList` and `ArrayList` classes had implemented the `List` interface directly, then they would have to implement all the abstract methods of `List` interface.

But in this case, the `AbstractList` class implements the methods of `List` interface and passes them on to `LinkedList` and `ArrayList`. So here we get the advantage of declaring type from the interface and abstract class flexibility of implementing the common behavior.

When To Use Abstract Class And Interface In Java

We mainly use an abstract class to define a default or common behavior of the child classes that will extend from this abstract class. An interface is used to define a contract between two systems that interact in an application.

Certain specific situations are ideal for interfaces to be used and certain problems that can be solved only using abstract classes. In this section, we will discuss when we can use the interface and when we can use abstract classes.

When to use an Interface:

- Interfaces are mainly used when we have a small concise functionality to implement.
- When we are implementing APIs and we know they won't change for a while, then that time we go for interfaces.
- Interfaces allow us to implement multiple inheritance. Hence when we need to implement multiple inheritance in our application, we go for interfaces.
- When we have a wide range of objects, again interfaces are a better choice.
- Also when we have to provide a common functionality to many unrelated classes, still interfaces are used.

When to use an Abstract Class:

- Abstract classes are mainly used when we need to use inheritance in our application.
- As interfaces deal with public methods and variables, whenever we want to use non-public access modifiers in our program, we use abstract classes.
- If new methods have to be added then it is better to do it in an abstract class than in interface. Because if we add a new method in the interface, the entire implementation changes as interfaces only have method prototypes and class implementation using the interface will provide the implementation.
- If we want different versions of the components being developed, then we go for abstract class. We can change abstract classes more easily. But interfaces cannot be changed. If we want a new version, then we have to write the entire interface again.
- When we want to provide a common implementation for all the components, then the abstract class is the best choice.

Interface Vs Abstract Class In Java

Given below are some of the differences between Interfaces and Abstract classes in Java.

Interface	Abstract class
An interface is declared using the 'interface' keyword.	An abstract class is declared using the 'abstract' keyword.
The interface can be implemented using the 'implements' keyword.	The abstract can be inherited using 'extends' keyword.
An interface cannot extend a class or implement an interface, it can only extend another interface.	An abstract class can extend a class or implement multiple interfaces.
Interface members can only be public.	Abstract class members can be public, private, or protected.
An interface cannot be used to provide an implementation. It can only be used as a declaration.	An abstract class can be used for implementing the interface.
Multiple inheritance can be achieved using interfaces.	Abstract class does not support multiple inheritance.
Interfaces support only static and final non-access modifiers.	Abstract supports all non-access modifiers like static, final, non-static, and non-final.
Interfaces can only have abstract methods. From Java 8, it can have static and default methods.	An abstract class can have an abstract or non-abstract method.

Enum Inheritance In Java

We have discussed enum data types in our discussion on data types in Java. All enums extend from `java.lang.Enum` class. This class `java.lang.Enum` is an abstract class.

Also, all enum classes in Java are 'final' by default. Hence, an attempt to inherit a class from any enum classes results in a compiler error.

As Java does not permit multiple inheritance, we cannot inherit enum class from any other class as enum class already inherits from java.lang.Enum. However, enum classes can implement interfaces in Java and this is called Enum inheritance in Java.

Given below is an example of Enum Inheritance in Java.

```
//WeekDays interface declaration
interface WeekDays {
    public void displaydays();
}
//enum class implementing WeekDays interface
enum Days implements WeekDays {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;

    public void displaydays() {           //Override displaydays method
        System.out.println("The day of the week: " + this);
    }
}

class Main {
    public static void main(String[] args) {
        Days.MONDAY.displaydays();        //access enum value
    }
}
```

Output:

```
The day of the week: MONDAY
```

Here we have an interface WeekDays with an abstract method prototype displaydays (). Then we define an enum class Days that implements WeekDays interface. Here we define enum values from SUNDAY to SATURDAY and also override the displaydays method.

Finally, in the main method, we access the enum value and display it.

Frequently Asked Questions

Q #1) What happens if you give a method body in the interface?

Answer: For Java versions before Java 8, the method's body is not allowed in the interface. But since Java 8, we can either define a default or static methods inside the interface.

Q #2) Can an interface have variables in Java 8?

Answer: We can have constant variables in Java 8 using static and final modifiers. But we cannot have instance variables in Java interfaces. Any attempt to declare instance variables in an interface will result in a compiler error.

Q #3) What are the improvements in interfaces in Java 8?

Answer: The most important improvement for interfaces in Java 8 is that static and default methods are allowed in interfaces. We can have methods declared as static or default and define them inside the interface.

Q #4) Can we override the default method in the Java interface?

Answer: No. It is not mandatory to override the default method in the interface. This is because when we implement an interface in a class, then the default method of the class is accessible to the implementation class. Hence using the object of the implementation class, we can access the default method of the interface.

Q #5) Can interfaces have fields in Java?

Answer: Yes, we can have fields or variables in interfaces in Java but by default, all these fields are static, final, and public.

Conclusion

In this tutorial, we have discussed the changes made to interfaces in Java 8. Java 8 introduced static and default methods in interfaces. Earlier we could have only abstract methods in the interface. But from Java 8 onwards, we can define default and static methods in Java.

Also, Java 8 allows the use of lambda expressions with the functional interfaces in Java. Then we also discussed abstract classes and interfaces and saw when to use each of them in Java. We have also seen the enum inheritance in Java.

We also discussed some of the differences between extends and implements, class and interface, abstract class and interface, etc.
