Java 8 - What is functional interface?

Java 8 - Is it possible to have a method definition in a functional interface or interface?

Java 8 - How do we qualify a method definition in a functional interface of interface?

Java 8 - Can I define a static method in an interface?

Which annotation is used to declare an interface as functional interface?

Java has forever remained an Object-Oriented Programming language. By object-oriented programming language, we can declare that everything present in the Java programming language rotates throughout the Objects, except for some of the primitive data types and primitive methods for integrity and simplicity. There are no solely functions present in a programming language called Java. Functions in the Java programming language are part of a class, and if someone wants to use them, they have to use the class or object of the class to call any function.

A **functional interface** is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface. ***A functional interface can have any number of default methods***. Runnable, ActionListener, Comparable are some of the examples of functional interfaces.

Functional Interface is additionally recognized as **Single Abstract Method Interfaces**. In short, they are also known as SAM interfaces. Functional interfaces in Java are the new feature that provides users with the approach of fundamental programming.

Functional interfaces are included in Java SE 8 with Lambda expressions and Method references in **order to make code more readable, clean, and straightforward**. Functional interfaces are interfaces that ensure that they include precisely only one abstract method. Functional interfaces are used and executed by representing the interface with an annotation called ***@FunctionalInterface***. As described earlier, functional interfaces can contain only one abstract method***. However, they can include any quantity of default and static methods.***

In Functional interfaces, there is no need to use the abstract keyword as it is optional to use the abstract keyword because, by default, the method defined inside the interface is abstract only. We can also call Lambda expressions as the instance of functional interface.

Before Java 8, we had to create anonymous inner class objects or implement these interfaces.

```
// Java program to demonstrate functional interface

class Test {
   public static void main(String args[])
   {
      // create anonymous inner class object
      new Thread(new Runnable() {
         @Override public void run()
         {
            System.out.println("New thread created");
         }
      }).start();
   }
}
```

Output

New thread created

Java 8 onwards, we can assign lambda expression to its functional interface object like this:

```
// Java program to demonstrate Implementation of
// functional interface using lambda expressions

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

```
        // lambda expression to create the object
        new Thread(() -> {
            System.out.println("New thread created");
        }).start();
    }
}
```

Output

New thread created

@FunctionalInterface Annotation

@FunctionalInterface annotation is used to ensure that the functional interface can't have more than one abstract method. In case more than one abstract methods are present, the compiler flags an 'Unexpected @FunctionalInterface annotation' message. However, it is not mandatory to use this annotation.

```
// Java program to demonstrate lambda expressions to
// implement a user defined functional interface.

@FunctionalInterface

interface Square {
    int calculate(int x);
}

class Test {
    public static void main(String args[])
    {
        int a = 5;

        // lambda expression to define the calculate method
        Square s = (int x) -> x * x;
```

```
        // parameter passed and return type must be
        // same as defined in the prototype
        int ans = s.calculate(a);
        System.out.println(ans);
    }
}
```

Output

25

Some Built-in Java Functional Interfaces

Since Java SE 1.8 onwards, there are many interfaces that are converted into functional interface. All these interfaces are annotated with @FunctionalInterface. These interfaces are as follows –

Runnable –> This interface only contains the **run()** method.

Comparable –> This interface only contains the **compareTo()** method.

ActionListener –> This interface only contains the **actionPerformed()** method.

Callable –> This interface only contains the **call()** method.

Java SE 8 included four main kinds of functional interfaces which can be applied in multiple situations. These are:

- **Consumer**
- **Predicate**
- **Function**
- **Supplier**

Amidst the previous four interfaces, the first three interfaces,i.e., Consumer, Predicate, and Function, likewise have additions that are provided beneath –

Consumer -> Bi-Consumer

Predicate -> Bi-Predicate

Function -> Bi-Function, Unary Operator, Binary Operator

1. Consumer

The consumer interface of the functional interface is the one that accepts **only one argument** or a **gentrified argument**. The consumer interface has **no return value**. **It returns nothing.** There are also

functional variants of the Consumer — DoubleConsumer, IntConsumer, and LongConsumer. These variants accept primitive values as arguments.

Other than these variants, there is also one more variant of the Consumer interface known as Bi-Consumer.

Bi-Consumer – Bi-Consumer is the most exciting variant of the Consumer interface. The consumer interface takes only one argument, but on the other side, the Bi-Consumer **interface takes two arguments**. Both, Consumer and Bi-Consumer have **no return value**. It also returns noting just like the Consumer interface. It is used in **iterating through the entries of the map.**

Syntax / Prototype of Consumer Functional Interface –

Consumer<Integer> consumer = (value) -> System.out.println(value);

This implementation of the Java Consumer functional interface prints the value passed as a parameter to the print statement. This implementation uses the Lambda function of Java.

2. Predicate

In scientific logic, a function that accepts an argument and, in return, generates a Boolean value as an answer is known as a predicate. Similarly, in the java programming language, a predicate functional interface of java is a type of function which accepts **a single value or argument** and does **some sort of processing on it**, and **returns a Boolean (True/ False) answer**. The implementation of the Predicate functional interface also encapsulates **the logic of filtering (a process that is used to filter stream components on the base of a provided predicate) in Java.**

Just like the Consumer functional interface, Predicate functional interface also has some extensions. These are IntPredicate, DoublePredicate, and LongPredicate. These types of predicate functional interfaces accept only primitive data types or values as arguments.

Bi-Predicate – Bi-Predicate is also an extension of the Predicate functional interface, which, instead of one, **takes two arguments,** does some processing, and **returns the boolean value.**

Syntax of Predicate Functional Interface –

public interface Predicate<T> {

```
    boolean test(T t);

}
```

The predicate functional interface can also be implemented using a class. The syntax for the implementation of predicate functional interface using a class is given below –

```
public class CheckForNull implements Predicate {

    @Override
    public boolean test(Object o) {

        return o != null;

    }
}
```

The Java predicate functional interface can also be implemented using Lambda expressions. The example of implementation of Predicate functional interface is given below –

```
Predicate predicate = (value) -> value != null;
```

This implementation of functional interfaces in Java using Java Lambda expressions is more manageable and effective than the one implemented using a class as both the implementations are doing the same work, i.e., returning the same output.

3. Function

A function is a type of functional interface in Java that receives **only a single argument and returns a value after the required processing**. There are many versions of Function interfaces because a primitive type can't imply a general type argument, so we need these versions of function interfaces. Many different versions of the function interfaces are instrumental and are commonly used in primitive types like double, int, long. The different sequences of these primitive types are also used in the argument.

These versions are:

Bi-Function – The Bi-Function is substantially related to a Function. Besides, it **takes two arguments, whereas Function accepts one argument.**

The prototype and syntax of Bi-Function is given below –

```
@FunctionalInterface
public interface BiFunction<T, U, R>
{

  R apply(T t, U u);
   .......

}
```

In the above code of interface, T, U are the inputs, and there is only one output that is R.

Unary Operator and Binary Operator – There are also two other functional interfaces which are named as Unary Operator and Binary Operator. They both extend the Function and Bi-Function, respectively. In simple words, Unary Operator extends Function, and Binary Operator extends Bi-Function.

The prototype of the Unary Operator and Binary Operator is given below –

1. Unary Operator

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, U>
{
   ……...
}
```

2. Binary Operator

```
@FunctionalInterface
```

public interface BinaryOperator<T> extends BiFunction<T, U, R>

{

……..

}

We can understand from the above example that the Unary Operator accepts only one argument and returns a single argument only. Still, in Unary Operator both the input and output values must be identical and of the same type.

On the other way, Binary Operator takes two values and returns one value comparable to Bi-Function but similarly like Unary Operator, the input and output value type must be identical and of the same type.

4. Supplier

The Supplier functional interface is also a type of functional interface that **does not take any input or argument and yet returns a single output.** This type of functional interface is generally used in the **lazy generation of values. Supplier functional interfaces are also used for defining the logic for the generation of any sequence**. For example – The logic behind the Fibonacci Series can be generated with the help of the Stream.generate method, which is implemented by the Supplier functional Interface.

The different extensions of the Supplier functional interface hold many other supplier functions like BooleanSupplier, DoubleSupplier, LongSupplier, and IntSupplier. The return type of all these further specializations is their corresponding primitives only.

Syntax / Prototype of Supplier Functional Interface is –

@FunctionalInterface

public interface Supplier<T>{

// gets a result

…….

// returns the specific result

……

```
T.get();

}

// A simple program to demonstrate the use
// of predicate interface

import java.util.*;
import java.util.function.Predicate;

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

      // create a list of strings
      List<String> names = Arrays.asList(
         "Geek", "GeeksQuiz", "g1", "QA", "Geek2");

      // declare the predicate type as string and use
      // lambda expression to create object
      Predicate<String> p = (s) -> s.startsWith("G");

      // Iterate through the list
      for (String st : names) {
         // call the test method
         if (p.test(st))
            System.out.println(st);
      }
   }
}
```

Output

Geek

GeeksQuiz

Geek2

Important Points/Observations:

Here are some significant points regarding Functional interfaces in Java:

In functional interfaces, there is only one abstract method supported. If the annotation of a functional interface, i.e., @FunctionalInterface is not implemented or written with a function interface, more than one abstract method can be declared inside it. However, in this situation with more than one functional interface, that interface will not be called a functional interface. It is called a non-functional interface.

There is no such need for the @FunctionalInterface annotation as it is voluntary only. This is written because it helps in checking the compiler level. Besides this, it is optional.

An infinite number of methods (whether static or default) can be added to the functional interface. In simple words, there is no limit to a functional interface containing static and default methods.

Overriding methods from the parent class do not violate the rules of a functional interface in Java.

The **java.util.function** package contains many built-in functional interfaces in Java 8.

Java 8 - can you create lambda expression for non-functional interfaces?

Lambda Expression

Objects are the base of java programming language and we can never have a function without an Object, that's why Java language provide support for using lambda expressions only with functional interfaces.

Lambda Expression are the way through which we can visualize functional programming in the java object oriented world. Objects are the base of java programming language and we can never have a function without an Object, that’s why Java language provide support for using lambda expressions only with functional interfaces. Since there is only one abstract function in the functional interfaces, there is no confusion in applying the lambda expression to the method. Lambda Expressions syntax is (argument) -> (body). Now let’s see how we can write above anonymous Runnable using lambda expression.

Java 8 - Can I add an abstract function in the interface annotated with @FunctionalInterface

NO we cannot add more abstract functions to the interface annotated with @FunctionalInterface. It throws compilation error. As the FI is SAM, adding more abstract methods to it makes it non function throwing compiling error.

In functional interfaces, there is only one abstract method supported. If the annotation of a functional interface, i.e., @FunctionalInterface is not implemented or written with a function interface, more than one abstract method can be declared inside it.

Java 8 - What is the difference between interface and abstract class in Java 8?

Difference between Abstract Class and Interface

abstract keyword is used to create an **abstract class** and it can be used with **methods** also whereas **interface** keyword is used to create **interface** and it **can't** be used with **methods**.

Subclasses use **extends** keyword to extend an **abstract** class and they need to provide **implementation** of all the declared methods in the abstract class unless the subclass is also an abstract class whereas subclasses use **implements** keyword to implement **interfaces** and should provide implementation for all the methods declared in the interface.

Abstract classes can have methods with implementation whereas interface provides absolute abstraction and can't have any method implementations. Note that from Java 8 onwards, we can create default and static methods in interface that contains the method implementations.

Abstract classes can have **constructors** but **interfaces can't** have **constructors**.

Abstract class have all the features of a normal java class except that we **can't instantiate** it. We can use abstract keyword to make a class abstract but **interfaces** are a completely different type and can have **only public static final constants** and **method declarations.**

Abstract classes methods can have access modifiers as public, private, protected, static but **interface** methods are **implicitly public and abstract**, we **can't** use any other access modifiers with interface methods.

A **subclass can extend only one abstract class** but it **can implement multiple interfaces.**

Abstract classes can extend **other class** and **implement interfaces but interface can only extend other interfaces.**

We can **run** an **abstract class** if it **has main()** method but we **can't run an interface** because they can't have main method implementation.

Interfaces are used to define contract for the subclasses whereas abstract class also define contract but it can provide other methods implementations for subclasses to use.

That's all for the difference between an interface and abstract classes, now we can move on to know when should we use Interface over Abstract class and vice versa.

Interface or Abstract Class

Whether to choose between Interface or abstract class for providing a contract for subclasses is a design decision and depends on many factors. Let's see when Interfaces are the best choice and when can we use abstract classes.

Java doesn't support multiple class level inheritance, so every class can extend only one superclass. But a class can implement multiple interfaces. So most of the times Interfaces are a good choice for providing the base for class hierarchy and contract. Also coding in terms of interfaces is one of the best practices for coding in java.

If there are a **lot of methods in the contract**, then abstract class is more useful because **we can provide** a **default implementation for some of the methods that are common for all the subclasses.** Also if subclasses **don't need** to **implement a particular method**, they can **avoid** providing the implementation **but in case of interface**, the subclass **will have to provide** the implementation for all the methods even **though it's of no use and implementation** is just empty block.

If our base contract keeps on changing then interfaces can cause issues because we can't declare additional methods to the interface without changing all the implementation classes, with the abstract class we can provide the default implementation and only change the implementation classes that are actually going to use the new methods.

Use Abstract classes and Interface both

Using interfaces and abstract classes together is the best approach to design a system. For example, in JDK java.util.List is an interface that contains a lot of methods, so there is an abstract class java.util.AbstractList that provides a skeletal implementation for all the methods of List interface so that any subclass can extend this class and implement only required methods. We should always start with an interface as the base and define methods that every subclass should implement and then if there are some methods that only certain subclass should implement, we can extend the base interface and create a new interface with those methods. The subclasses will have the option to choose between the base interface and the child interface to implement according to its requirements. If the number of methods grows a lot, it's not a bad idea to provide a skeletal abstract class implementing the child interface and providing flexibility to the subclasses to choose between interface and an abstract class.

Java 8 interface changes

From Java 8 onwards, we can have method implementations in the interfaces. We can create default as well as static methods in the interfaces and provide an implementation for them. This has bridged

the gap between abstract classes and interfaces and now interfaces are the way to go because we can extend it further by providing default implementations for new methods

What is the default access modifier (accessibility or scope of a field, method, constructor, or class) for methods declared in an interface?

What is the default access modifier for fields and variables?

PUBLIC ABSTRACT - methods

PUBLIC STATIC FINAL - fields

Shall we declare static methods in an interface? If so, how do we do? What is the purpose?

Java interface static method is similar to default method except that **we can't override them** in the implementation classes. This feature helps us in avoiding undesired results in case of poor implementation in implementation classes.

Static Methods in Interface are those methods, which are defined in the interface with the keyword static. Unlike other methods in Interface, these static methods contain the complete definition of the function and since the definition is complete and the method is static, therefore these methods cannot be overridden or changed in the implementation class.

Similar to Default Method in Interface, the static method in an interface can be defined in the interface, but cannot be overridden in Implementation Classes. To use a static method, Interface name should be instantiated with it, as it is a part of the Interface only.

Important points about java interface static method:

Java interface static method is part of interface, we can't use it for implementation class objects.

Java interface static methods are good for providing utility methods, for example null check, collection sorting etc.

Java interface static method helps us in providing security by not allowing implementation classes to override them.

We can't define interface static method for Object class methods, we will get compiler error as "This static method cannot hide the instance method from Object". This is because it's not allowed in java, since Object is the base class for all the classes and we can't have one class level static method and another instance method with same signature.

We can use java interface static methods to remove utility classes such as Collections and move all of it's static methods to the corresponding interface, that would be easy to find and use.

```
package com.journaldev.java8.staticmethod;


public interface MyData {


	default void print(String str) {

		if (!isNull(str))

			System.out.println("MyData Print::" + str);

	}


	static boolean isNull(String str) {

		System.out.println("Interface Null Check");


		return str == null ? true : "".equals(str) ? true :
false;

	}

}
```

Java interface static method is visible to interface methods only, if we remove the isNull() method from the MyDataImpl class, we won't be able to use it for the MyDataImpl object. However like other static methods, we can use interface static methods using class name. For example, a valid statement will be:

```
boolean result = MyData.isNull("abc");
```

Does Java 9 support multiple inheritance? If so, how is it implemented?

No java does not support multiple inheritance with classes. But we can make use of the interface to achieve multiple inheritance through interfaces. With the additional features of Java 8 and

introduction of default and static methods to the interface concept , multiple inheritance is achievable to a little extent.

Can I declare an object with super type (parent class) while creating a sub type (child class)?

No. It makes zero sense to allow that. The reason is because subclasses generally define additional behaviour. If you could assign a superclass object to a subclass reference, you would run into problems at runtime when you try to access class members that don't actually exist.

Using an object of super (parent) class, can I call a method defined in the sub (child) class?

No, a superclass has no knowledge of its subclasses. Yes, a subclass has access to all nonprivate members of its superclass.

Can I declare an object of sub (child) class and assign an object of super (parent) class?

Can we assign subclass object to superclass?

Assigning subclass object to a superclass variable

Therefore, if you assign an object of the subclass to the reference variable of the superclass then the subclass object is converted into the type of superclass and this process is termed as widening (in terms of references)

Converting one data type to others in Java is known as casting.

If you convert a higher datatype to lower datatype, it is known as narrowing (assigning higher data type value to the lower data type variable).

char ch = (char)5;

If you convert a lower data type to a higher data type, it is known as widening (assigning lower data type value to the higher data type variable).

Int i = 'c';

Similarly, you can also cast/convert an object of one class type to others. But these two classes should be in an inheritance relation. Then,

If you convert a Super class to subclass type it is known as narrowing in terms of references (Subclass reference variable holding an object of the superclass).

Sub sub = (Sub)new Super();

If you convert a Sub class to Super class type it is known as widening in terms of references (super class reference variable holding an object of the sub class).

Super sup = new Sub();

Therefore, if you assign an object of the subclass to the reference variable of the superclass then the subclass object is converted into the type of superclass and this process is termed as widening (in terms of references).

But, using this reference you can access the members of superclass only if you try to access the subclass members a compile-time error will be generated.

Which keyword is used to represent the parent class in the child class while accessing methods or fields defined in the parent class?

Super

We can use **super keyword** to access the data member or field of parent class. It is used if parent class and child class have same fields.

Which access modifiers declared in the parent class are inherited to the child class if parent and child classes are in 2 different packages?

Public and protected. – different packages.

Public , protected , private – same package

Does constructor inheritable?

Constructors are not members, so **they are not inherited by subclasses**, but the constructor of the superclass can be invoked from the subclass.