

What is use of Functional Interface in Java 8?

[Ask Question](#)

I came across a new term named **Functional Interface** in Java 8.

I could only find one use of this interface while working with *lambda expressions*.

Java 8 provides some built-in functional interfaces and if we want to define any functional interface then we can make use of the `@FunctionalInterface` annotation. It will allow us to declare only a single method in the interface.

For example:


```
@FunctionalInterface
interface MathOperation {
    int operation(int a, int b);
}
```

How useful it is in Java 8 other than just working with *lambda expressions*?

The question [here](#) is different from the one I asked. It is asking why we need *Functional Interface* while working with Lambda expression. My question is: why using *Functional Interface* other than directly with *lambda expressions*?

[java](#) [java-8](#)

edited May 10 at 23:34

 [pkpnd](#)
2,836 7 30

asked Apr 27 '16 at 6:21

 [Madhusudan](#)
1,349 5 23 54

1 It looks duplicate to this link. They also talk about why there should be only one method in Functional Interface: stackoverflow.com/questions/33010594/... – [Kulbhushan Singh](#) Apr 27 '16 at 6:29

1 @KulbhushanSingh I saw this question before posting... Both questions sense difference... – [Madhusudan](#) Apr 27 '16 at 6:32

check out a detailed explanation on functional interfaces and changes on interfaces of java 8: sysdotoutdotprint.com/index.php/2017/04/22/java-8-interface – [me13kings](#) Apr 22 '17 at 0:36

6 Answers

`@FunctionalInterface` annotation is useful for compilation time checking of your code. You cannot have more than one method besides `static`, `default` and abstract methods that override methods in Object in your `@FunctionalInterface` or any other interface used as a functional interface.

But you can use lambdas without this annotation as well as you can override methods without `@Override` annotation.

From docs

a functional interface has exactly one abstract method. Since default methods have an implementation, they are not abstract. If an interface declares an abstract method overriding one of the public methods of `java.lang.Object`, that also does not count toward the interface's abstract method count since any implementation of the interface will have an implementation from `java.lang.Object` or elsewhere

This **can be used** in lambda expression:

```
public interface Foo {
    public void doSomething();
}
```

This **cannot be used** in lambda expression:

```
public interface Foo {
    public void doSomething();
    public void doSomethingElse();
}
```

But this will give **compilation error**:

```
@FunctionalInterface
public interface Foo {
    public void doSomething();
}
```

```
public void doSomethingElse();
}
```

Invalid '@FunctionalInterface' annotation; Foo is not a functional interface

edited Nov 3 '17 at 4:20



Andrea Bergonzo

352 1 3 16

answered Apr 27 '16 at 6:30



Sergii Bishyr

3,853 3 16 33

- 18 To be more precise, you have to have *exactly one* abstract method that doesn't override a method in `java.lang.Object` in a functional interface. – [Holger](#) Apr 27 '16 at 8:04
- 1 @Holger Yeah, that's exactly what the docs says. – [Sergii Bishyr](#) Apr 27 '16 at 8:07
- 6 ...and it's slightly different to "not have more than one public method besides static and default "... – [Holger](#) Apr 27 '16 at 8:12
- 2 @Holger you are right. I have added quote from docs to my answer. Thanx! :) – [Sergii Bishyr](#) Apr 27 '16 at 8:16
- 1 @SergheyBishyr Thanks That's awesome explanation (Y) – [Girdhar Singh Rathore](#) May 17 '17 at 6:52

Functional interfaces have a single functionality to exhibit. For example, a `Comparable` interface with a single method `'compareTo'` is used for comparison purpose. Java 8 has defined a lot of functional interfaces **to be used extensively in lambda expressions**.

There's an annotation introduced- `@FunctionalInterface` which can be used for compiler level errors when the interface you have annotated is not a valid Functional Interface.

```
@FunctionalInterface
interface MathOperation {
    int operation(int a, int b);
}
```

Let's try to add another abstract method:

```
@FunctionalInterface
interface MathOperation {
    int operation(int a, int b);
    int operationMultiply(int a, int b);
}
```

Above will result into compiler error as given below:

```
Unexpected @FunctionalInterface annotation
@FunctionalInterface ^ MathOperation is not a functional interface
multiple non-overriding abstract methods found in interface MathOperation
```

A functional interface is valid even if the `@FunctionalInterface` annotation would be omitted. It is only for informing the compiler to enforce single abstract method inside interface.

```
interface MathOperation {
    int operation(int a, int b);
}
```

Conceptually, a functional interface has exactly one abstract method. Since default methods have an implementation, they are not abstract. Since default methods are not abstract you're **free to add default methods to your functional interface as many as you like**.

Below is valid functional interface:

```
@FunctionalInterface
interface MathOperation {
    int operation(int a, int b);
    default void doSomeMathOperation(){
        //Method body
    }
}
```

If an interface declares an **abstract method overriding one of the public methods of `java.lang.Object`**, that also does not count toward the interface's abstract method count since any implementation of the interface will have an implementation from `java.lang.Object` or elsewhere.

e.g. Below is a valid functional interface even though it declared two abstract methods. Why? Because one of these abstract methods `"equals()"` which has signature equal to public method in `Object` class.

```
@FunctionalInterface
interface MathOperation {
    int operation(int a, int b);
}
```

```

@Override
public String toString(); //Overridden from Object class
@Override
public boolean equals(Object obj); //Overridden from Object class
}

```

While the intended use of **Functional interfaces** is for **lambda expressions**, **method references** and **constructor references**, they can still be used, like any interface, with anonymous classes, implemented by classes, or created by factory methods.

edited Apr 27 '16 at 14:35



Hank D

3,796 2 12 27

answered Apr 27 '16 at 6:32



Prakash Hari Sharma

1,143 2 14 20

- 1 All interface methods are implicitly public. The `public` specifier is not necessary and even discouraged. – Devashish Jaiswal Mar 22 at 5:27

The [documentation](#) makes indeed a difference between the purpose

An informative annotation type used to indicate that an interface type declaration is intended to be a *functional interface* as defined by the Java Language Specification.

and the use case

Note that instances of functional interfaces can be created with lambda expressions, method references, or constructor references.

whose wording does not preclude other use cases in general. Since the primary purpose is to indicate a *functional interface*, your actual question boils down to “*Are there other use cases for functional interfaces other than lambda expressions and method/constructor references?*”

Since *functional interface* is a Java language construct defined by the Java Language Specification, only that specification can answer that question:

[JLS §9.8. Functional Interfaces](#):

...

In addition to the usual process of creating an interface instance by declaring and instantiating a class (§15.9), instances of functional interfaces can be created with method reference expressions and lambda expressions (§15.13, §15.27).

So the Java Language Specification doesn't say otherwise, the only use case mentioned in that section is that of creating interface instances with method reference expressions and lambda expressions. (This includes constructor references as they are noted as one form of method reference expression in the specification).

So in one sentence, no, there is no other use case for it in Java 8.

answered Apr 27 '16 at 8:34



Holger

140k 17 183 361

Not at all. Lambda expressions are the one and only point of that annotation.

answered Apr 27 '16 at 6:25



Louis Wasserman

138k 17 237 311

- 4 Well, lambdas work without the annotation as well. It's an assertion just like `@Override` to let the compiler know that you intended to write something that was "functional" (and get an error if you slipped). – Thilo Apr 27 '16 at 6:34
- 1 Straight to the point and the correct answer, though a bit short. I took the time to add a [more elaborated answer](#) saying the same thing with more words... – Holger Apr 27 '16 at 8:41

A lambda expression can be assigned to a functional interface type, but so can method references, and anonymous classes.

One nice thing about the specific functional interfaces in `java.util.function` is that they can be composed to create new functions (like `Function.andThen` and `Function.compose`, `Predicate.and`,

etc.) due to the handy default methods they contain.

answered Apr 27 '16 at 6:46



[Hank D](#)

3,796

2

12

27

You should elaborate on this comment more. What about method references and new functions? – [K.Nicholas](#)
Mar 14 at 22:13

As others have said, a functional interface is an interface which exposes one method. It may have more than one method, but all of the others must have a default implementation. The reason it's called a "functional interface" is because it effectively acts as a function. Since you can pass interfaces as parameters, it means that functions are now "first-class citizens" like in functional programming languages. This has many benefits, and you'll see them quite a lot when using the Stream API. Of course, lambda expressions are the main obvious use for them.

answered Apr 27 '16 at 14:39



[Sina Madani](#)

507

5

18