Why do I need a functional Interface to work with lambdas?

Ask Question

I think this question is already somewhere out there, but I wasn't able to find it.

I don't understand, why it's necessary to have a functional interface to work with lambdas. Consider the following example:

```
public class Test {

    public static void main(String...args) {
        TestInterface i = () -> System.out.println("Hans");
//      i = (String a) -> System.out.println(a);

        i.hans();
//      i.hans("Hello");
    }
}

public interface TestInterface {
    public void hans();
//  public void hans(String a);
}
```

This works without problems, but if you uncomment the commented lines, it doesn't. Why? In my understanding, the compiler should be able to distinct both methods, since they have different input-parameters. Why do I need a functional interface and blow up my code?

EDIT: The linked duplicates didn't answer my question because I'm asking about different method-parameters. But I got some really useful answers here, thanks to everyone who helped! :)

EDIT2: Sorry, I'm obviously not a native speaker, but to precise myself:

```
public interface TestInterface {
    public void hans();                 //has no input parameters</br>
    public void hans(String a);         //has 1 input parameter, type String</br>
    public void hans(String a, int b);  //has 2 input parameters, 1. type =
String, 2. type = int</br>
    public void hans(int a, int b);     //has also 2 input parameters, but not
the same and a different order than `hans(String a, int a);`, so you could
distinguish both
}

public class Test {

    public static void main(String...args) {
        TestInterface i = () -> System.out.println("Hans");
        i = (String a) -> System.out.println(a);
        i = (String a, int b) -> System.out.println(a + b);
        i = (int a, int b) -> System.out.println(a);

        i.hans(2, 3);   //Which method would be called? Of course the one that
would take 2 integer arguments. :)
    }
}
```

All I'm asking is about the arguments. The method name doesn't matter, but each method takes an unique order of different arguments and because of that, Oracle could have implemented this feature instead just making a single method possible per "Lambda-Interface".

java lambda java-8 functional-interface

edited Oct 8 '15 at 11:33

asked Oct 8 '15 at 8:34

codepleb
2,900 6 27 52

3 `int i = 7; i = 5; System.out.println(i); // wouldn't it be awesome if this would print both 7 and 5?` – immibis Oct 8 '15 at 9:40

1 I really don't understand your reasoning. If something doesn't implement an interface, then how can it be of that type? An iPod doesn't count as iPhone just because you don't happen to make phone calls. – zeroflagL Oct 8 '15 at 10:44

@immibis: What you describe here is different from what I asked. :) – codepleb Oct 8 '15 at 10:47

2 @TrudleR You seemed to be asking: `<some type> i = <some value>; i = <some other value>; // why can't i be both values at once here?` The variable `i` can hold a reference to ***one*** object, no matter whether that's an object created by a lambda expression or an instance of a class you wrote explicitly. – immibis Oct 8 '15 at 23:19

1 @TrudleR So you want to do something like `TestInterface i = new TestInterface() {void hans() {System.out.println("Hans");} void hans(int a) {System.out.println(a);} /* and so on */};` ? – immibis Oct 9 '15 at 11:17

4 Answers

When you write :

```
TestInterface i = () -> System.out.println("Hans");
```

You give an implementation to the `void hans()` method of the `TestInterface` .

If you could assign a lambda expression to an interface having more than one abstract method (i.e. a non functional interface), the lambda expression could only implement one of the methods, leaving the other methods unimplemented.

You can't solve it by assigning two lambda expressions having different signatures to the same variable (Just like you can't assign references of two objects to a single variable and expect that variable to refer to both objects at once).

answered Oct 8 '15 at 8:40

Eran
249k 32 379 466

2 @TrudleR, then Java would be a functional language. If you are interested in such implementation on JVM. You could try out the Scala language. – Damian Leszczyński - Vash Oct 8 '15 at 8:50

4 @TrudleR: “if you could just call methods for which you implemented a body”—how do you know which methods are implemented when you have an instance of an arbitrary interface implementation? – Holger Oct 8 '15 at 9:24

1 TrudleR, you got me thinking here :) but IMHO they'd be opening a Pandora box if they allow to "just call methods for which you implemented a body"...: lambda defines 1 method,while your interface is a contract for 2; deviating from this contract would really confuse the receive of your object.. plus if your interface has several methods with a String parameter, the compiler won't know which one is matched by the lambda expression. If you want to go radical, you could consider node.js which simply doesn't bother with contracts(=interfaces)... – Pelit Mamani Oct 8 '15 at 9:53

1 @TrudleR: I don’t get your point. Surely, neither anonymous classes nor interfaces are obsolete, nobody said something different. That doesn’t justify your request of being able to implement a multi-method interface by defining *one* valid method and all others being broken. I ask again, how is a programmer (or compiler) seeing a reference of that interface type (which may hold an arbitrary implementation) supposed to know which method is the one that is not broken? I’m not talking about the place where you define your `n-1` broken implementation but the place where the object is being used. – Holger Oct 8 '15 at 11:05

1 In your question, you are assigning the variable `i` twice (if you uncomment the lines). Just to get your idea correctly… you declare a variable of type `TestInterface` and initialize it with a lambda expression which ought to mean something like “create an incomplete implementation of `TestInterface` with only one method”, then you assign the variable a different lambda expression which ought to mean “merge this lambda expression into the implementation class behind that reference, so that now both methods are defined”? Seriously? – Holger Oct 8 '15 at 11:23

|

The most important reason why they must contain only one method, is that confusion is easily possible otherwise. If multiple methods were allowed in the interface, which method should a lambda pick if the argument lists are the same ?

```
interface TestInterface {
    void first();
    void second(); // this is only distinguished from first() by method name
    String third(); // maybe you could say in this instance "well the return type
is different"
    Object fourth(); // but a String is an Object, too !
}

void test() {
    // which method are you implementing, first or second ?
    TestInterface a = () -> System.out.println("Ido mein ado mein");
    // which method are you implementing, third or fourth ?
    TestInterface b = () -> return "Ido mein ado mein";
}
```

answered Oct 8 '15 at 8:51

blagae
1,549 1 15 35

Yeah I know that, it should just result in an Exception if you want to run such code. But I thought that different arguments should be clear for the compiler. If you declare 2 methods with the same name and parameters, it also gives an exception, so why should Oracle prevent it here? – codepleb Oct 8 '15 at 9:04

Good example. Thanks – greenhorn Nov 14 '17 at 9:15

You seem to be looking for **anonymous classes**. The following code works:

```
public class Test {

    public static void main(String...args) {
        TestInterface i = new TestInterface() {
            public void hans() {
                System.out.println("Hans");
            }
            public void hans(String a) {
                System.out.println(a);
            }
        };

        i.hans();
        i.hans("Hello");
    }
}

public interface TestInterface {
    public void hans();
    public void hans(String a);
}
```

Lambda expressions are (mostly) a shorter way to write anonymous classes with only one method. (Likewise, anonymous classes are shorthand for inner classes that you only use in one place)

answered Oct 9 '15 at 11:20

immibis
32.7k 4 34 62

Hey I'm not that much of a newbie. :) Of course I know how I can bring it to work. I just wanted to ask, why lambdas are cut to only work with functional interfaces. It's clear that I would need to write anonymous classes (or implement a class itself) for my case. All I wanted to say is, that I think Oracle could have extended the use of lambdas. But after the discussions here it's clear why they force you use functional interfaces. But thanks anyways. – codepleb Oct 9 '15 at 11:50

@TrudleR If they extended lambdas in the way you're thinking of, then they'd be the same as anonymous classes, so what would be the point? – immibis Jan 12 '17 at 20:11

precise answer thanks. – greenhorn Nov 14 '17 at 9:13

You do not have to create a functional interface in order to create lambda function. The interface allow you to create instance for future function invocation.

In your case you could use already existing interface Runable

```
Runnable r = () -> System.out.println("Hans");
```

and then call

```
r.run();
```

You can think of lambda `->` as only short hand for:

```
Runnable r = new Runnable() {
     void run() {
          System.out.println("Hans");`
     }
}
```

With lambda you do not need the anonymous class, that is created under the hood in above example.

But this has some limitation, in order to figure out what method should be called interface used with lambdas must be SAM (Single Abstract Method). Then we have only one method.

For more detailed explanation read:

Introduction to Functional Interfaces – A Concept Recreated in Java 8

edited Oct 8 '15 at 8:44

answered Oct 8 '15 at 8:37

Damian Leszczyński - Vash
24k 7 44 86