Quiz Yourself: Functional Interfaces (Advanced)

JAVA SE

Quiz Yourself: Functional Interfaces (Advanced)

Define and implement functional interfaces that work as expected.

by Simon Roberts and Mikalai Zaikin

February 27, 2020

If you have worked on our quiz questions in the past, you know none of them is easy. They model the difficult questions from certification examinations. The “intermediate” and “advanced” designations refer to the exams rather than to the questions, although in almost all cases, “advanced” questions will be harder. We write questions for the certification exams, and we intend that the same rules apply: Take words at their face value and trust that the questions are not intended to deceive you but to straightforwardly test your knowledge of the ins and outs of the language.

The objective here is to test your knowledge of how to write functional interfaces.

Given the four interfaces:

```
interface Printable {
    void print();
}

interface Stringable {
    String toString();
}

interface Cloneable {
    Object clone();
}

interface InfoItem extends Printable, Stringable, C
```

Which are valid functional interfaces in Java 11? Choose two.

A. `Printable`
B. `Stringable`
C. `Cloneable`
D. `InfoItem`

Answer. The question investigates the rules governing functional interfaces. All the interfaces are valid from the simple perspective that they compile; that is, the syntax of each is a valid Java interface. However, functional interfaces have additional constraints, which are described in the *Java Language Specification* for Java 11 in section 9.8.

The simple way that these rules are usually presented is to say that a functional interface has exactly one abstract method. This is often referred to as a *single abstract method* or *SAM* interface. This terminology is actually core to the specifications of some other languages.

Given this, you can see that options A, B, and C look possible, but option D looks less so. The interface defined in option D would require the methods `print()`, `toString()`, and `clone()`. However, you probably recognize the signatures of the `toString()` and `clone()` methods. Both of these actually exist in the class `java.lang.Object`. Given that they're defined there, they must exist in any other object too. Does this mean they're not really abstract methods?

In fact, the Java specification addresses this question by defining an additional restriction:

> The definition of functional interface excludes methods in an interface that are also public methods in `Object`.

From this, it should be clear that the `toString()` method, which takes no arguments and returns `String`, is exactly the same `toString()` method defined as a `public` member of `Object`. Because of this, the `Stringable` interface is not a valid Java functional interface, because it defines zero abstract methods that qualify under the rule above. This tells you that `Stringable` is not a functional interface and option B is incorrect.

So, does this mean that option C is also incorrect, because `Object` includes a `clone()` method that takes zero arguments? It does not. The catch is that the `clone()` method in `Object` is declared as `protected`. It looks like this:

```
protected native Object clone() throws CloneNotSupp
```

Here again, the question is whether the method defined in the `Cloneable` interface in option C qualifies for the single abstract method necessary for a functional interface. The rules are clear; it doesn't matter that a `clone()` method appears in `Object`; because `clone()` is not `public` in `Object`, `Cloneable` is considered to be a functional interface, and option C is correct.

Option A should be the easy one. The `print()` method is the only abstract method (indeed, the only method) declared in the interface and it is not present in `Object`. From this, you can conclude that option A is also correct.

The interface in option D is a little different. It extends three other interfaces. An `abstract` method in an interface constitutes an obligation to implement that method. When one interface extends another, unless it adds a `default` method that satisfies one of those obligations, the effect is that all the obligations from all the extended interfaces are aggregated in the new interface. So, in the example below, all three interfaces simply declare the single abstract method `doIt()`, and all are valid functional interfaces:

```
@FunctionalInterface
interface A {
    void doIt();
}

@FunctionalInterface
```

```
interface B extends A {}

@FunctionalInterface
interface C extends B {}
```

Given this, you can see that the interface in option D requires two `abstract` methods: `print()` and `clone()`. Remember that the `toString()` method does not qualify as being abstract in the terms of this discussion, so it doesn't get counted here. Because there are two `abstract` methods, this interface breaks the most basic requirement that there must be exactly one. Therefore, `InfoItem` cannot be a functional interface and option D is incorrect.

As a side note, keep in mind that the interface `java.lang.Cloneable` is entirely distinct from the `Cloneable` interface introduced in this question. The interface `java.lang.Cloneable` has no methods and is, therefore, not a functional interface. It is called a *marker interface*, and this term is used for some other zero-method interfaces, including `java.io.Serializable`. A marker interface is a way of labeling a class so that a runtime decision can be made about instances that implement that interface; this approach was used before Java 5 introduced annotations to the language. The `clone()` method declared in `Object` exists regardless of whether a subclass declares the `java.lang.Cloneable` interface.

The correct answers are options A and C.

Simon Roberts

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who publishes recorded and live video training through Pearson InformIT (available direct and through the O'Reilly Safari Books Online service). He remains involved with Oracle's Java certification projects.

Mikalai Zaikin

Mikalai Zaikin is a lead Java developer at IBA IT Park in Minsk, Belarus. During his career, he has helped Oracle with development of Java certification exams, and he has been a technical reviewer of several Java certification books, including three editions of the famous *Sun Certified Programmer for Java* study guides by Kathy Sierra and Bert Bates.

Share this Page