Quiz Yourself: Default Methods (Advanced)

What fields can you access from default methods in interfaces?

by Simon Roberts and Mikalai Zaikin

January 10, 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.

Given the code:

```java
interface Nameable {
    default void setName(String name) {
        this.name = name;
    }
    default String getName() {
        return this.name;
    }
}

class Employee implements Nameable {
    protected String name;
}

class HR {
    public static void main(String[] args) {
        Employee e = new Employee();
        e.setName("John Doe");
        System.out.println(e.getName());
    }
}
```

**What is the result?** Choose one.

A. Interface Nameable fails to compile.
B. Class Employee fails to compile.
C. Class HR fails to compile.
D. John Doe is output.

**Answer.** This code investigates aspects of the default methods feature added to Java 8. Default methods are instance methods that are defined, with implementations, in an interface. Although there are differences in how such methods are inherited compared to regular instance methods defined in classes, this feature nevertheless creates a form of multiple implementation inheritance in Java. Multiple implementation inheritance was not included in Java when it was first created because this language feature caused a lot of design issues in code written in C++. 

```java
interface Nameable {
    default void setName(String name) {
        this.name = name;
    }
    default String getName() {
        return this.name;
    }
}

class Employee implements Nameable {
    protected String name;
}

class HR {
    public static void main(String[] args) {
        Employee e = new Employee();
        e.setName("John Doe");
        System.out.println(e.getName());
    }
}
```
To limit the problems caused by multiple inheritance, Java takes two steps. The first is simply an admonition to use the feature for specific and limited purposes, notably the extension of library interfaces (although it’s interesting that the core APIs themselves actually break this guidance). The second is that although methods may be defined, instance variables may not (other than `public static final` values, which has always been possible). Also, fields declared in classes are not visible to interfaces (remember that classes implement interfaces, but interfaces have no way to know which classes might implement them); consequently, no instance variables are ever visible to the `default` methods. Because of this, the really troublesome problems of multiple implementation inheritance, notably the “diamond inheritance” problem, are avoided.

As a consequence of these design choices, although `default` methods do have a `this` reference (they are instance methods), the only things that can be accessed through the reference are the instance methods (`abstract` and `default`) and `public static final` fields declared in the interface. It’s not possible to refer directly to any regular instance state directly. (Of course, the implementation of an `abstract` method might do so, but such code is written in a class, not in the interface.)

In this question, there is no field called `name` declared in the `Nameable` interface. Consequently, the attempt to use `this.name` in the implementations of the two `default` methods in the interface `Nameable` cannot compile. From this, you know that option A is correct.

Let’s look at the question of putting a variable in the interface. Suppose you add `String name` to the `Nameable` interface like this:

```java
interface Nameable {
    String name = "John Doe";
    ...
}
```

By default, all fields in the interface are `final` (hence, the assignment is mandatory during declaration), and they are `public` and `static`.

With this addition, the `getName` method that was declared in the `Nameable` interface will compile. Here it is again for convenience:

```java
default String getName() {
    return this.name; // OK
}
```

However, the `setName` method will still fail to compile because the `final` variable `name` may not be reassigned:

```java
default void setName(String name) {
    this.name = name; // NOT OK
}
```

Therefore, even with the addition of a `String name` static field in the interface, the code still won’t compile as shown without making further changes to the code of the `default` methods.

Given that the `Nameable` interface fails to compile, it’s not entirely clear whether the other classes could compile. Generally, however, if you have a question of this kind, where one part of the code definitely fails to compile because of a particular error in that code, and other parts of the given code are not in themselves in error, and you are expected to pick exactly one answer describing code that fails to compile, it’s safe to assume that the correct answer is the one related to the overt error. In fact, in this case, the classes `Employee` and `HR` are correct in all respects other than depending on the erroneous `Nameable`. But of
There is actually a little more to discuss about the String name static field that was proposed earlier. First, it would be possible to have a mutable element if a StringBuilder were used instead of a String. But it would be the content of the object that is being changed, not the reference value stored in name. Another consideration is that, as mentioned, this is a static field; there is only one name value shared among all the instances of all the classes that implement Nameable. This would probably be a surprising effect given the use of instance getter and setter methods.

The correct answer is option A.