Quiz yourself: Working with abstract classes and default methods in Java

JAVA SE

Quiz yourself: Working with abstract classes and default methods in Java

What happens when a default method is hidden and inaccessible?

by Mikalai Zaikin and Simon Roberts

March 2, 2021

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. 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 following classes and an interface:

```
interface Flyable {
    public default void fly() {
        System.out.print("Default fly");
    };
}

abstract class Bird implements Flyable {
    public abstract void fly();
}

class Chicken extends Bird implements Flyable
    public void fly() {
        System.out.print("Cannot fly");
    };
}

class BirdsFarm {
    public static void main(String[] args) {
        Bird b = new Chicken();
        Flyable f = b;
        f.fly();
        }
    }
```

What is the result? Choose one.

A. The `Bird` class fails to compile. The answer is A.

B. The `Chicken` class fails to compile. The answer is B.

C. `Default fly` is printed. The answer is C.

D. `Cannot fly` is printed. The answer is D.

Answer. The code in this question sets out three types in a hierarchy: the interface `Flyable`, the abstract class `Bird`, and the concrete class `Chicken`. The interface `Flyable` provides a default method `fly()`.

The `Bird` class is perhaps a little surprising. First, it hides the default `fly` method, replacing it with an abstract method of the same signature. This is where the surprise might be: The default method is now inaccessible, and any subclasses must explicitly provide a new implementation of `fly`. Although it might be surprising, it *is* valid in Java, and the code compiles. Therefore, option A is incorrect. It might also be surprising that subclasses cannot use the `super` keyword to get at the default `fly` method defined in `Flyable`.

Next, notice that the `Chicken` class implements the `fly` method. As just described, it is actually mandatory for the `Chicken` to implement `fly` if `Chicken` is to be a concrete class. The syntax is valid and the code compiles successfully. So, the `Chicken` class is valid, the code compiles successfully, and option B is incorrect.

There are a couple of additional things to note for this part of the discussion. First, even if the `Bird` class did not hide the `Flyable.fly` method, thereby preventing access to it from the `Chicken` class, that would not be a problem. Why? Because you should expect overriding methods to replace parent methods whether they're concrete methods in classes or the default methods in an interface. Although it's redundant, explicitly stating that the `Chicken` class `implements Flyable` is legal syntax. (However, it does not in any way reveal the hidden `fly` method in the `Flyable` interface.)

The second thing to note is that if a class implements two interfaces that have default methods with the same signatures, and neither that class nor the classes it extends explicitly implement those methods, then having two default methods *does* create a conflict. In such a case, the compiler refuses to make a judgment about which of the two default methods should be used, and compilation fails.

By the way, this is a form of the *diamond inheritance problem*. (See Michael Kölling's article, "The evolving nature of Java interfaces," and also another one of our quizzes.)

However, if there is a concrete implementation of the default methods anywhere in the class hierarchy (and here we're being specific about *classes* as opposed to *interfaces*), then that method resolves the conflict. Such a method can usually use a variant of the `super` syntax to invoke a selected default method from an interface. In a typical case, that syntax might look like `Flyable.super.fly()` except that, as has been noted, in this particular example, the abstract version of `fly` completely hides the default one.

We've now established that the code compiles. When it runs, the normal rules apply to the selection of the `fly` method that will be executed. This decision about which to execute is made according to the type of the object (which is a `Chicken`), not according to the type of the reference (which is `Flyable`). This is the normal late or virtual binding behavior for nonprivate instance methods in Java. As a result, the output will be `Cannot fly`, which makes option D correct and option C incorrect.

By the way, Simon's three chickens, Dutchess, Sheila, and Buffy, would like you to know that they are offended by this question, asserting strenuously that they *can* fly, albeit not very well.

Conclusion: The correct answer is option D.

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.

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.

Share this Page

Contact

US Sales: +1.800.633.0738
Global Contacts
Support Directory
Subscribe to Emails

About Us

Careers
Communities
Company Information
Social Responsibility Emails

Downloads and Trials

Java for Developers
Java Runtime Download
Software Downloads
Try Oracle Cloud

News and Events

Acquisitions
Blogs
Events
Newsroom

ORACLE Integrated Cloud
Applications & Platform Services

© Oracle | Site Map | Terms of Use & Privacy | Cookie Preferences | Ad Choices