JAVA SE

Quiz yourself: Abstract methods of concrete classes in Java

All the abstract methods inherited by a concrete class must have concrete implementations, or the code cannot compile.

by Simon Roberts and Mikalai Zaikin

July 26, 2021 | Download a PDF of this article
More quiz questions available here

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 code

```
interface Text {
    default String getContent() { return "Blank";
    void setContent(String txt);
    void spellCheck() throws Exception;
}

abstract class Prose {
    public abstract void setAuthor(String name);
    public void spellCheck() {
        System.out.print("Do generic prose spellch
    }
}

class Novel extends Prose implements Text {
  // line n1
}
```

Which fragments added simultaneously at line n1 allow the following two lines of code to compile and run? Choose two.

```
Novel n = new Novel();
n.spellCheck();
```

A. `public void spellCheck() throws Exception { }` — The answer is A.

B. `String getContent() { return "Novel"; }` — The answer is B.

C. `public String getContent() { return "Novel"; }` — The answer is C.

D. `public void setAuthor(String a) { }` — The answer is D.

E. `public void setContent(String txt) { }` — The answer is E.

Answer. The class `Novel` does not carry the modifier `abstract`. It is, therefore, a *concrete class*, and this requires that all the abstract methods it inherits, whether directly or indirectly, must have concrete implementations or the code cannot compile.

All the methods defined in an interface (in Java 8 or later) are public, and they are abstract, default, or static. Any method that is not explicitly marked as either default or static is abstract, so this means that the `Text` interface declares two abstract methods, which are `setContent` and `spellCheck`. These two methods, therefore, must be implemented before the `Novel` class can satisfy the requirements for being concrete.

The interface, additionally, provides a default method called `getContent`. Although `getContent` is not considered a concrete method, it is not abstract either. It is acceptable for a class that claims to be a concrete implementation of this interface to have no mention of this method.

At this point, you know that `Novel` must still provide or obtain concrete implementations for `setContent` and `spellCheck`—and that these implementations must provide the correct argument lists.

Let's move on and consider what the abstract class `Prose` brings to `Novel`.

`Prose` declares one abstract method, `setAuthor`. The `Novel` class must obtain or provide a concrete implementation for that method. At this point, therefore, you are looking for concrete implementations of three methods to satisfy the needs of `Novel`.

However, `Prose` also provides a concrete implementation of `spellCheck`, and the argument list exactly matches the same-named abstract method in the interface. Therefore, this method satisfactorily resolves the requirement that the `Novel` class obtain or provide an implementation of the abstract `spellCheck` method declared in the interface.

This information leaves you with a need for two concrete methods to be implemented in the `Novel` class, and you know those must be `setAuthor` and `setContent`. These are the two lines of code in options D and E; therefore options D and E are correct.

Let's explore why the other options are incorrect. Start by considering why option A is incorrect. The interface declares an abstract method `spellCheck` that throws `Exception` in its signature. Is the `spellCheck` defined in the `Prose` class sufficient for this? It turns out that, yes, it is sufficient. It's completely OK for an implementation of an interface method to be inherited from elsewhere in the class's hierarchy, so the method can come from a parent class or from a default method in another interface.

A related question is whether a method that does not throw exceptions can satisfy a requirement for a method that does. The answer to this, too, is yes. Simply because a method declares an exception does not mean it will throw the exception; it means only that it *might*. Something that never actually throws the exception is OK. *The converse, however, is not OK*. A concrete method that declares checked exceptions that are not declared (either exactly or using a superclass) by an abstract method cannot correctly provide the implementation of that abstract method.

Option B will not compile. The method that it attempts to override is declared (and given a default implementation) in the interface, and even though no access modifier is provided, it is implicitly public. An overriding method may not reduce the accessibility of the method it overrides. Given that the method declared in option B has default access, which is less accessible than public access, compilation fails. Therefore, option B is incorrect.

(By the way, from an exam-taking perspective, code doesn't always have to compile correctly. So even if the code compiled, selecting option B would "use up" one of the two options that you can select. Because both options D and E are necessary, if you chose option B, you'd end up missing one of the options that are required.)

Finally, option C presents code that would compile but is not necessary. The method `getContent` is already provided by the default implementation in the interface. And as with the discussion of option B, if you selected option C, that would prevent you from selecting option D or option E. Therefore, you should not select option C and it is incorrect.

Conclusion. The correct answers are options D and E.

Related quizzes

- Quiz yourself: Define the structure of a Java class
- Quiz yourself: Working with abstract classes and default methods in Java
- Quiz yourself: Create and extend abstract classes

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