JAVA SE

Quiz yourself: Create and extend abstract classes

Test your knowledge of abstract classes and their methods.

by Simon Roberts and Mikalai Zaikin

November 16, 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. 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 of this Java SE 11 quiz is to create and extend abstract classes.

Which of the following abstract classes will fail to compile? Choose one.

A.

```
interface Intf1 {}
abstract class Cls1 implements Intf1 {  }
```

The answer is A.

B.

```
interface Intf2 {
  void m2();
}
abstract class Cls2 implements Intf2 {
  public void m2() {}
}
```

The answer is B.

C.

```
interface Intf3 {}
abstract class Cls3 implements Intf3 {
  void m3();}
```

The answer is C.

D.

```
interface Intf4 {
  default void m4() {};
}
abstract class Cls4 implements Intf4 {
  public abstract void m4();
}
```

The answer is D.

Answer. There are several important rules about abstract classes, for example:

- An abstract class can never be directly instantiated (though a concrete subclass can be).
- An abstract class may contain abstract methods (but is not required to).
- A concrete class must not contain abstract methods.
- Abstract methods in a parent class or interface remain abstract in a subclass or implementation unless explicitly provided with a concrete implementation.
- An abstract method must not be given a body, but it must end with a semicolon.
- A nonabstract method must have a body.

Option A declares an abstract class that implements an interface, and neither the class nor the interface declares any methods (abstract or otherwise). The result is an abstract class without any abstract methods. From the rules above, you can see that this is entirely legal, and of course you could remove the `abstract` modifier from the `Cls1` declaration and it would still compile successfully. From this you know that the code in option A compiles. Therefore, option A is incorrect.

Option B declares an abstract class that provides concrete implementations of all the abstract methods from `Intf2`, and so it has no abstract methods. As with option A, the class can be `abstract` without having any abstract methods. In addition, as with option A, you could remove the `abstract` modifier from

the `Cls2` declaration, and the resulting code would compile successfully. Therefore, option B is incorrect.

It's perhaps worth pondering why you might be allowed to mark a class `abstract` if it contains no abstract methods. The essence of this is that being abstract primarily means the class cannot be instantiated directly, and that's often because the class represents an abstraction that does not physically exist in the real world.

For example, you could have an abstract class called `Animal` with concrete subclasses such as `Lion`, `Tiger`, and `Dog`. It makes sense to say `new Dog()`, because you know what to expect: four legs, a wagging tail, and exuberant enthusiasm for your company when you get home from work. But saying `new Animal()` wouldn't make sense because you don't know what type of animal should be created. "Animal" is an idea, a category, or a generalization perhaps, but it is not an actual thing you can instantiate in this context.

So, having classes that can't be instantiated makes good sense in its own right. But notice that none of that discussion touched on whether the `Animal` class needed any abstract methods.

The other side of the coin is that an abstract method constitutes an unfulfilled promise. If you could make an object that doesn't fully implement all the expected methods, that would cause problems at runtime. Consequently, you are permitted to declare abstract methods only in types (abstract classes and interfaces) that cannot be directly instantiated, and any derived concrete type must provide a concrete implementation for those methods.

In option C, there is no intrinsic problem with declaring the abstract class `Cls3` and asserting that it implements `Intf3`. However, the `m3()` method has no body; that is, it ends with a semicolon and not with curly braces bounding a method body. This would be fine if the method were marked `abstract`. In this case, though, it is not marked `abstract`, and that combination of nonabstract method and absence of a method body is not legal, so compilation would fail. Because of this, option C is correct.

Option D addresses an interesting variation not discussed in the initial list of rules. It turns out that you can redeclare a concrete method as an abstract one in a subclass. Given this, the `default` nonabstract `m4()` method from `Intf4` can be correctly redeclared as `abstract` in the `Cls4` class, though of course the `Cls4` class must be declared `abstract` too.

By the way, because the declaration of the `abstract` version of method `m4` is in a class, you must explicitly mark it as `public`. Otherwise it will take on package-level accessibility, which is the default in Java, and that would mean less accessibility than provided by the implicitly `public` version declared in the interface. When you override a method (even when you are not

actually implementing it), making the method less accessible than the version already declared is prohibited because it would violate the Liskov substitution principle.

Conclusion: The correct answer is option 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