Quiz

Quiz yourself: Sealed and non-sealed classes and interfaces

December 12, 2022 | 5 minute read


Simon Roberts


Mikalai Zaikin

Sealed types limit the set of types that can be assigned to a base type.

More quiz questions available here

Given the following code

```
sealed interface Super permits Sub1, Sub2, Sub3 {}
non-sealed class Sub1 implements Super {}
record Sub2(Super s) implements Super {}
enum Sub3 implements Super { SUB_A{}, SUB_B{} }
```

Which statement is true? Choose one.

A. The code compiles as it is. The answer is A.

B. The record Sub2 is invalid. The answer is B.

C. The enum Sub3 is invalid. The answer is C.

D. Both record Sub2 and enum Sub3 are invalid. The answer is D.

Answer. The purpose of `sealed` types is to let you limit the set of types that can be assignment-compatible with a particular base type.

A common example of inheritance would be a base type `Animal` with child classes for `Lion`, `Tiger`, `Bear`, and so on. This structure makes perfect sense if you decide to add `Snake`, `Dog`, `Bird`, and any number of additional types that are subtypes of `Animal`.

Sometimes, however, you might need to model a situation where there are a limited number of

possible or permissible subtypes. An example would be a base type `Quadrilateral` (four-sided geometric shape). The possible subtypes of this would be `Square`, `Rectangle`, `Parallelogram`, `Trapezium`, `Trapezoid`, `Rhombus`, and `Kite`. (Let's not get picky about the geometric definitions, as we recall that two of those have different meanings on opposite sides of the Atlantic Ocean!)

There are good reasons (some of which are related to the form of `switch/case` construction that's a preview feature in Java 17 and Java 18) why it can be helpful to the integrity of the design to be able to enforce this restriction and not allow any additional subtypes. It's beyond this article's scope to discuss those reasons, but it's sufficient to know that the `sealed` class syntax is provided to allow you to enforce this.

To make the `sealed` class syntax work, the syntax must allow you to define a base type (which can be an interface, abstract class, or concrete class) and enumerate the permitted subtypes. Further, the syntax must enforce controls on the subtypes of those subtypes. This goal suggests the following rules:

> A `sealed` type must (generally—but this article will not get into the exception) explicitly declare the permitted subtypes.
>
> Each permitted subtype must either be `sealed` or `final` (in which case it cannot have any subtypes).
>
> It's also permitted to have a subtype that's explicitly declared as `non-sealed`, in which case it allows any number of subtypes.

All `sealed` classes have some interactions with existing type rules. For example, `abstract` classes and interfaces cannot be `final`, so that possibility is removed and either of them can be `sealed` or `non-sealed`, but they cannot be `final`.

You can probably guess that enums, which prohibit arbitrary subclassing, and `record` types, which are implicitly `final`, also interact with these rules. That will be explored shortly.

Now that you've got a good starting point, consider the elements presented in this question.

The declaration of `Super` is valid. An interface can be `sealed`, the syntax is good, and the declaration enumerates the three permitted subtypes: `Sub1`, `Sub2`, and `Sub3`.

The declaration of the class `Sub1` is also valid. It demonstrates the use of the `non-sealed` idea. It's permitted to have arbitrary subtypes, and it's explicit about that. Thus it is correct.

The `record Sub2` declaration is also valid. It's not labeled `final`, `sealed`, or `non-sealed`, but a record is implicitly `final`, so it's acceptable in that respect. Also, although a record cannot declare a parent class, it's fine for a record to implement an interface, as this one does. You could add the `final` modifier explicitly, but that is not required, so the code for `Sub2` compiles.

The `Sub3` enum is interesting. Unlike the record `Sub1`, an enum is not necessarily `final`. It's true that you can't use `extends` from an enum, but you can create subclasses if you do so inside the enum itself—and the code in the question does just that.

Both the constant values `SUB_A` and `SUB_B` are followed with curly brace pairs, and this means that each instance is a subclass of the `Sub3` enum. Because the subtypes of an enum are so tightly controlled, the *Java Language Specification* treats an enum that contains children in this way as being *implicitly* `sealed`. Section 8.9 of the specification for Java 17 says

> An enum class is either implicitly `final` or implicitly `sealed`, as follows:

> An `enum` class is implicitly `final` if its declaration contains no `enum` constants that have a class body.
>
> An `enum` class E is implicitly `sealed` if its declaration contains at least one `enum` constant that has a `class` body. The permitted direct subclasses of E are the anonymous classes implicitly declared by the `enum` constants that have a `class` body.

In the situation where no `class` bodies were provided for any of the individual `enum` constants, the `enum` class would be *implicitly* `final` and would work too.

Because the `Sub1` class is explicitly marked `non-sealed`, the record `Sub2` is implicitly `final`, and the enum `Sub3` is implicitly `sealed`, all three are valid and the syntax of the code presented is entirely correct and compiles. In view of that, you can see that option A is correct, and therefore options B, C, and D must be incorrect.

Conclusion. The correct answer is option A.

Related quizzes

Quiz yourself: Sealed class true-or-false questions

Quiz yourself: The effects of declaring a Java class as final

Quiz yourself: Enums and implementing interfaces in Java

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
Show more

Mikalai Zaikin

Mikalai Zaikin is a lead Java developer at IBA Lithuania (part of worldwide IBA Group) and currently located in Vilnius. During his career, Zaikin has helped Oracle with development of Java certification exams, and he has been a technical reviewer of several Java certification books, including three
Show more

Previous Post

Nothing is better than the Optional type. Really. Nothing is better.

Michael Ernst | 9 min read

Next Post

Quiz yourself: Lambda expressions and local variables in Java

Mikalai Zaikin | 5 min read