Quiz Yourself: Final Classes
(Intermediate)

The effects of declaring a class final

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.

Which of the following is a benefit of declaring a Java class as final? Choose one.

A. It promotes the principle “prefer inheritance over composition.”
B. It simplifies class unit testing.
C. It helps to design immutable classes.
D. It prevents instantiation of the class.

Answer. When the final modifier is applied to a class, it prevents that class from being subclassed. From the syntax perspective, it’s as simple as that. Of course, syntax exists to provide some benefit for the design, and the programmer must weigh the design options in the context of the problem being solved. Let's consider in which cases a final class might be a benefit.

In his highly regarded book Effective Java, Joshua Bloch makes a compelling argument for preferring composition over inheritance when you want to reuse the code of one class in another, and this principle is widely accepted in the canon of today’s object-oriented design wisdom. Option A, however, suggests the exact opposite approach. The option suggests that inheritance is a good thing, even though current wisdom states the opposite. This should make you severely doubt that option A is the correct choice. But, because final in this context actually prevents subclassing, it prevents the use of inheritance and couldn’t be correct even if inheritance were preferable to composition. Therefore, you know that option A is incorrect.

To illustrate the distinction between inheritance and composition, consider this final class A:

```java
final class A {  
    Result doUsefulStuff(DomainData dd) { /* implementation */
}
```

The following code, which attempts to use inheritance, will not compile:
For class B to succeed in reusing the code of instance methods embedded in class A, it must have an explicit reference to an object of type A, which can be available as method arguments or as a field in class B. Using a field would qualify as composition (or perhaps aggregation; the difference is not significant to this particular discussion), and the code would look something like this:

```java
class B {
    A support;
    void performCalculation() {
        DomainData dd = // initialize
        // intermediate calculations
        Result r = support.doUsefulStuff(dd);
        // more calculations
    }
}
```

Option B suggests that a final class might be easier to test. At first glance, you might consider that inheritance creates additional complexity, because it allows behaviors to be combined and that might make it hard to be confident that all possible combinations have been fully tested. Of course, the same can be said for combinations formed using the composition approach discussed in the context of option A. So overall, complexity issues don’t give a clear general reason to favor one approach over the other.

From another perspective, however, there’s a case to be made for suggesting that preventing inheritance might make some approaches to testing actually harder. A common approach to testing is to create mock objects, and a fairly obvious approach to this is to create a subclass and override the methods that need to be changed. Of course, a final class would prevent you from doing this directly. However, it’s also true that modern mocking libraries are generally able to hack their way around this issue, and you could argue that a design that emphasizes the use of pure functions would likely reduce the need for mocking anyway.

Overall, it should be clear that marking classes final is not likely to have any reliable impact on testing, and to the extent it does, it’s likely to be detrimental because it makes testing harder. From this, you conclude that option B is incorrect.

Option C suggests that immutable classes offer design benefits and that marking a class final will help with immutability. Let’s consider whether immutable classes might be beneficial first. If you use mutable classes (which has been the usual approach for most object-oriented software for a long time), any part of the code that receives a copy of the reference to an object can change that object’s state. This can create some very surprising situations, and “surprising situations” is really a euphemism for “bug that’s hard to track down.” A simple, but illustrative, example might be a piece of software that manages student registrations for a school. Imagine that the heart of the system maintains a list of all the students and a junior programmer is asked to write a piece of code that prints a list of all the students who are “in good standing”; this might mean they’ve paid all their bills, no assignments are overdue, and they have a passing grade. The programmer has been told to write a method that receives a list of students as an argument. He writes the method, but implements it by deleting the students who are not in good standing and printing the result. The code is demonstrated, and the output is deemed correct.
imagine that the method isn’t used very often, but some time after it’s been used, someone tries to send messages to all the students and quite a while later, someone notices that the master list is incomplete. When a problem is found by its consequences, it’s typically hard to debug.

By contrast, if the original list had been immutable, this error could not have arisen. So, although the discussion is much more complex than can be addressed here, with luck you can see that immutable data can offer design benefits. The next question would be whether applying final to a class can help make it immutable.

The answer is yes, it can, and it’s almost essential. If you build a class that has private fields, does not provide any mutator methods, doesn’t store in its own state any mutable objects provided to it, and doesn’t return any of its own potentially mutable objects to callers, you have a good start at immutability. However, if the class permits subclassing, the subclass can redefine the behaviors in such a way as to permit mutability. This is the case even if the original fields were private, because new variables that hide the original ones can be created and mutated like this:

```java
// Immutable parent
class Parent {
    private int value;
    public int getValue() { return value; } // Getter
}

// Mutable, but assignment-compatible child
class Child extends Parent {
    private int value; // shadowing variable
    public int getValue() { return value; } // Getter
    public void setValue(int v) { value = v; } // Setter
}
```

By making a class final, you avoid the risk that a child class will be created and will break some semantic contract that was intended for your class. Therefore, option C is correct.

Option D is incorrect; it suggests that final restricts instantiation, but that is not the case. The only restriction is that the class cannot be extended; final has no influence on instantiation and such a class can have as many instances as desired unless this is restricted in some other way. To protect a class from instantiation, a programmer can make the class abstract (which does not prevent instantiation of any subclasses and actually prohibits marking the class final) or mark all of the constructors of the class private and never create instances in the methods of the class. Some limited use cases for such classes do exist. The java.lang.Math class is such an example, although a singleton type approach is likely preferable in most cases for modern designs.

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.