JAVA SE

Quiz yourself: Refactoring with the core functional interfaces (intermediate)

See if you know how to improve code by refactoring.

by Simon Roberts and Mikalai Zaikin

June 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. 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.

The objective in this Java SE 11 quiz is to improve code through the use of refactoring. Suppose that looking through the code commits history, you notice that your colleague has replaced a method signature from this:

```
public void testDrive(Vehicle x) {
  // ...
}
```

To this:

```
public void testDrive(Supplier<? extends Vehi
  // ...
}
```

Which of the following is an advantage of such refactoring?
Choose one.

A. Faster execution	The answer is A.
B. Better encapsulation	The answer is B.
C. Lazy instantiation	The answer is C.
D. Supporting the caching of vehicles	The answer is D.
E. Facilitating the use of different vehicle subclasses	The answer is E.

Answer. When a question's answer choices seem to be value judgments or perhaps opinion-based, it's always tempting to get distracted by considerations of whether the question is rigorous and how you're supposed to know what the examiner was thinking. However, you don't need to be a mind reader.

You should feel confident that the real exam questions went through extensive critique, and significant beta testing, to weed out questions (and answers) that were open to reasonable disagreement. In this case, only we two authors wrote the question, so there's far less discussion and critique. However, we hope to illustrate how you can answer such a question through reason and logic, even when you're busy contorting yourself thinking, "Well, maybe, if I look at it *this* way," when considering the options.

Let's think about the options in turn.

Option A suggests the resulting code will run faster, so you need to consider how the two options might compare. In the original code, it's the responsibility of the caller to provide the `Vehicle` object directly. In the refactored form, the caller provides a `Supplier`, which is a means to access a `Vehicle`. However, the `Supplier` is not constrained to create a `Vehicle` either; it could simply return an existing one.

Now, it could be that the caller already has the `Vehicle` object for other reasons (perhaps many tests are being done in sequence on the same `Vehicle`). Both the old and refactored case can work with an existing object, but in the refactored case, the `testDrive` method is forced to go through a layer of indirection to access the `Vehicle`. So far, there's no obvious benefit to the refactored approach, but there's a potential downside to it.

Another consideration is what happens if the `testDrive` method doesn't actually need the `Vehicle` to do its work. That might seem like a bit of a stretch, given the name of the method, but it could be that the test itself determines it should skip execution, perhaps because of a configuration flag. If that were the case, the refactored form can simply not use the `Supplier`. This would potentially allow for no `Vehicle` to be made at all, and that might give faster performance.

At this point, there's a bit of an argument for performance in both directions, depending on context that you know nothing about, so this option does not look very compelling.

Option B suggests better encapsulation results from the refactored form. This seems unreasonable. In either case, you have `Vehicle` objects, and whether those are encapsulated or not depends on their implementation, not on their use. In both cases, the `testDrive` method knows about the `Vehicle`; it's only the access mechanism that has changed. And as with the `Vehicle` itself, there's really no impact on the encapsulation or otherwise of the `testDrive` method and the class that it belongs to. Thus, encapsulation seems like a pretty solid no. Therefore, you can deduce that option B is incorrect.

Option C talks about lazy instantiation. What does that mean? Well, this is, in part, the idea referred to in option A. It means that the called method gets to control *when*, and even *whether*, to create the `Vehicle`. While it's hard to imagine test-driving a vehicle without a vehicle, we suggested earlier that the test might be controlled by some configuration flag and sometimes simply skip its own execution. If that situation arose, this approach supports such skipping directly and efficiently. But lazy instantiation actually goes further; it is possible to imagine a test-drive needing more than one vehicle (perhaps the first one gets damaged in a crash!). After receiving a `Supplier`, the called method can invoke that supplier zero, one, or *multiple* times, and thereby create exactly the number of objects it requires.

A question: This approach effectively gives the called method control over instantiation, but does it, therefore, take control away from the caller? Might that be a counteracting disadvantage that would force you to be as critical of this option as of option A? The answer here is no; the caller still gets to control everything about *how* the `Vehicle` is created; it only loses some control over *when* the object is created. In fact, the caller can still provide a `Supplier` that merely provides access to an existing object, rather than allowing new ones to be created, though in this case, there needs to be some coordination of design intentions between the programmers of the caller and the called method to ensure the two aspects aren't working to defeat one another.

At this point, it should be clear that the potential benefit described in option A is simply a subset of the benefit in option C. Given that the question requires a single answer and option B is incorrect, this should lead you to decide option A is incorrect and to look at option C as the only contender at this point.

Option D suggests that the refactored approach supports caching of the `Vehicle` instance. Certainly, it's true that an implementation of the `Supplier` can interact with a cache. But the caller is responsible for providing the `Supplier`, so it's the caller's responsibility to ensure that this is addressed. In the prerefactoring design, the caller provides the `Vehicle` directly

and can just as easily access the cache itself. On balance, then, while it's true that `Supplier`s can interact with caches under the caller's control, this cannot be seen as a specific advantage, since the old approach also allows this, and neither approach seems to do that better or worse than the other. Therefore, option D is incorrect.

Option E suggests that the `Supplier` will facilitate the use of different subclasses of `Vehicle`. This option is clearly nonsensical. Yes, the notion of generalization is what allows a subclass of `Vehicle` to be substituted at runtime. In this case, the caller is responsible for supplying a `Vehicle`, or a `Supplier` of `Vehicle`, and in either case, the caller has the option to ensure that the called method ends up with any assignment-compatible object. Therefore, option E is incorrect.

From these discussions, we hope we've made a case that shows option C is clearly better than option A, and that the other three options do not present any real benefit. Therefore, option C is the correct answer.

As a side note, the potential benefits of lazy instantiation can be very significant in the right situations, and the approach is widely used now in logging frameworks. It's common for the preparation of a log message to involve walking up stack frames in an `Exception` and performing a lot of string concatenation. All of this can consume significant CPU and perhaps memory too. And yet, it's also quite common for the message to be abandoned entirely if, for example, it's a "trace" level message, but the logging level is set to "error" or "severe." By passing a `Supplier<String>` to the logging system, instead of passing the actual `String`, all that preparatory computation can be avoided unless the message will actually be used.

In older versions, the code for a simple logging call might have looked like this:

```
logger.log(Level.FINER,
  Stream
    .of(t.getStackTrace())
    .map(StackTraceElement::toString)
    .collect(Collectors.joining("\n"));
```

Notice that in this case, the message has been created before that staggeringly expensive call to `log` is made, even though the logger might simply throw the message away.

Alternatively, the following form acknowledges that the message construction operation might never be needed if the logging level indicates the message is not needed. Of course, this manual stack walking and collection is deliberately done manually here to emphasize the point; it's not intended to represent a good way to process your exception traces.

```
logger.log(Level.FINER,
  () -> Stream
    .of(t.getStackTrace())
    .map(StackTraceElement::toString)
    .collect(Collectors.joining("\n"));
```

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