JAVA SE

Quiz yourself: Determining eligibility for garbage collection in Java

You never can tell when the Java garbage collector will pick up the trash.

by Mikalai Zaikin and Simon Roberts

April 5, 2021

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.

For this quiz, you will dig into garbage collection. Given the following classes:

```
abstract class Test {
    JavaClass classToTest;
    abstract public void doTest(JavaClass cla
}
class UnitTest extends Test {
    JavaClass classToUnitTest;
    public UnitTest(JavaClass classToTest) {
        this.classToTest = classToTest;
    }
    @Override
    public void doTest(JavaClass classToTest)
        classToUnitTest = classToTest;
    }
}
class JavaClass {}
```

and a fragment of a test case scenario:

```
JavaClass c1 = new JavaClass(); // line A
Test t1 = new UnitTest(c1);
t1.doTest(c1);
JavaClass c2 = c1;      // line 1
c1 = new JavaClass();   // line 2
c2 = null;              // line 3
t1.doTest(null);        // line 4
System.gc();            // line 5
t1 = new UnitTest(c1); // line 6
```

After which line's execution will the object created at line A become eligible for garbage collection? Choose one.

A. Line 1 — The answer is A.

B. Line 2 — The answer is B.

C. Line 3 — The answer is C.

D. Line 4 — The answer is D.

E. Line 5 — The answer is E.

F. Line 6 — The answer is F.

G. Not possible to predict, because it depends on the garbage collector (GC) implementation — The answer is G.

Answer. Before you start reviewing the answer options, you should understand the question's requirements.

An object becomes eligible for garbage collection when it can't be reached directly or indirectly from any live thread. Threads can access objects starting from method-local reference variables at any point in the active region of that thread's stack or from any static fields. In the question's sample code, only a single user thread is shown, and there aren't any static fields. This means you need to be concerned only with tracing reachability starting from method-local variables shown in the code.

Further, it's important to distinguish an *object eligible for garbage collection* from *garbage actually collected*. The former can be evaluated accurately, but actual garbage collection is out of your control. In fact, it's possible that garbage collection could simply never recover *any* objects during the life of a program. How? That could happen if very little memory is allocated so there's never any pressure to recover memory. It could also happen if the Epsilon GC is in use, which never reclaims anything. The Epsilon GC might seem strange, but it can be used for evaluating the memory behavior of a program.

Before line 1, there are two local variables: `c1` and `t1`. The `JavaClass` instance created at line A (let's call this the *target object*) is reachable directly from the `c1` reference. It's also reachable indirectly by following the `t1` reference to the `UnitTest` object, and from there through either of the two fields

inside that object. Both `classToTest` and `classToUnitTest` refer to that target object. (See **Figure 1**.)

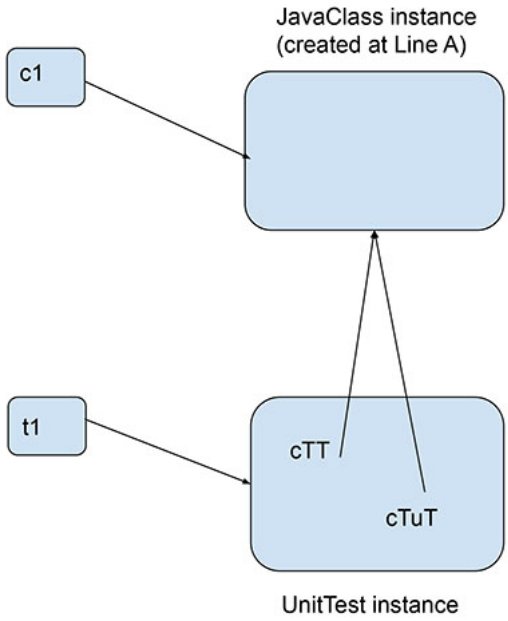

Figure 1. The JavaClass and UnitTest instances before line 1

With three paths to reach the target object, clearly it cannot be collected as garbage.

Line 1 creates another local variable, `c2`, and assigns that to refer to the target object. There are now four ways to reach the target object, so clearly it cannot be collected as garbage at this point, and thus option A is incorrect.

Line 2 overwrites `c1` with a reference to a new instance of `JavaClass`. The target object is still reachable via `t1` (which contains two links to the object) and `c2`. The situation now looks like **Figure 2**.

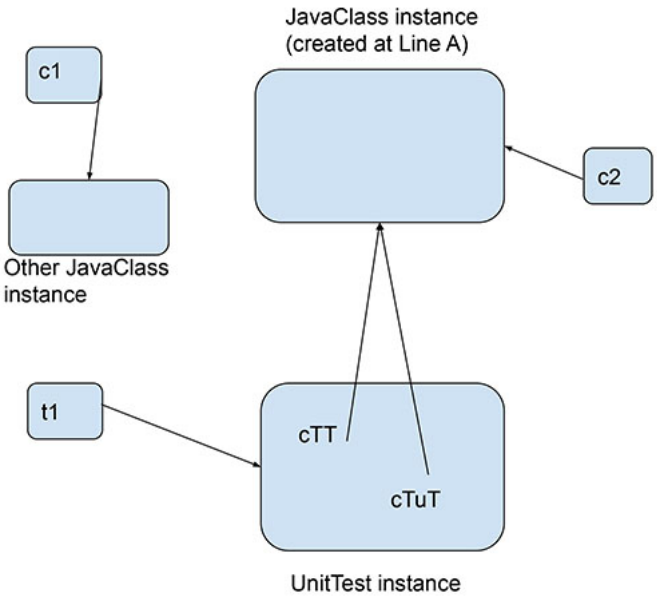

Figure 2. The instances after line 2

Line 3 overwrites the `c2` variable with `null`, but the target object is still reachable through the two fields of `t1`. So, option C is incorrect.

At line 4, the field `t1.classToUnitTest` is overwritten with a `null` pointer, but the other field, `t1.classToTest`, still maintains a reference, as seen in **Figure 3**. Therefore, the target object is still reachable and cannot be garbage collected.

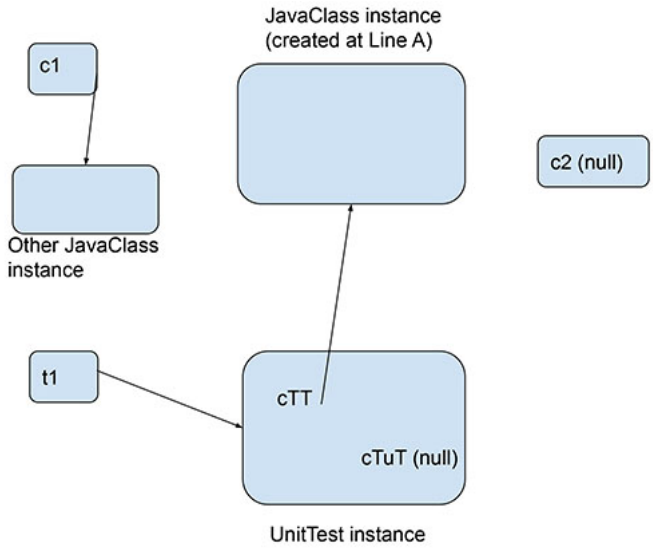

Figure 3. The instances after line 4

The target object is still reachable, and so option D is incorrect.

Option E is a simple distraction. The call to `System.gc()` suggests that it might be beneficial if the GC runs, but this call has no effect on what is *eligible* for collection. In fact, the call might not even cause the GC to run. This method is considered a hint, but the actual decision about garbage is made by the JVM—in fact, the hint can be entirely ignored. This means option E must be incorrect.

After line 6, `t1` has been reassigned, and because of this, there is no way to reach the target object from the local variables that are in scope, as shown in **Figure 4**.

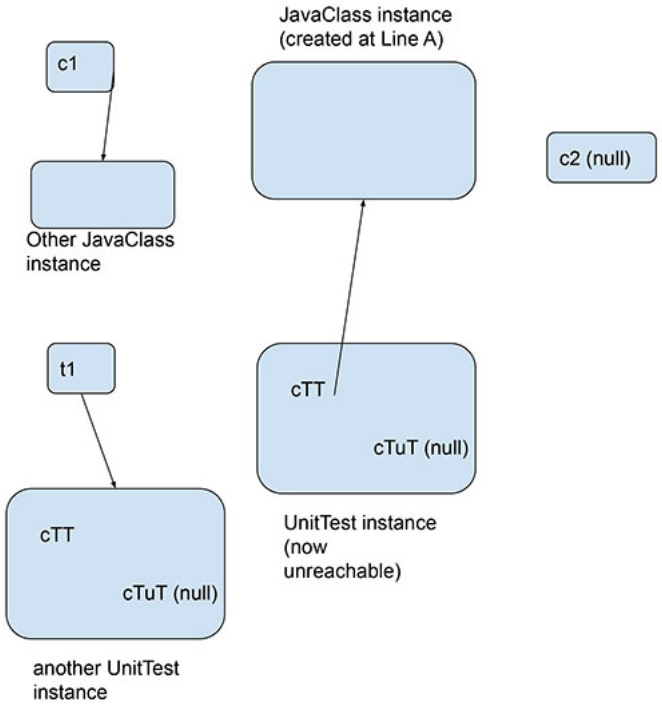

Figure 4. The instances after line 6

Even though the `classToTest` field in the original `UnitTest` instance refers to the target object, there's no way to reach that `UnitTest` object; consequently, there's no way to reach the target object from any reference available to the thread. Therefore, the target object is eligible for garbage collection. This tells you that option F is correct.

Option G is incorrect. As discussed in relation to option E, you can never predict the *execution* of the GC process. You can only know when an object becomes *eligible* for collection.

Conclusion: The correct option is F, because the `JavaClass` instance created at line A will be eligible for garbage collection after line 6 has been completed.

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.

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.

Share this Page