Quiz yourself: When are Java objects eligible for garbage collection?

A pirate treasure can be found as long as there’s some map or reference. So too with a Java object.

by Mikalai Zaikin and Simon Roberts

May 31, 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.

This quiz is about your understanding of an object’s lifecycle, especially garbage collection. Given the following class

```java
public class GCDemo {
    public static ArrayList<Object> l = new ArrayList<>();
    public void doIt() {
        HashMap<String, Object> m = new HashMap<>();
        Object o1 = new Object(); // line n1
        Object o2 = new Object();
        m.put("o1", o1);
        o1 = o2; // line n2
        o1 = null; // line n3
        l.add(m);
        m = null; // line n4
        System.gc(); // line n5
    }
}
```

And given this code fragment
When does the object created at line n1 become eligible for garbage collection? Choose one.

A. At line n2

B. At line n3

C. At line n4

D. At line n5

E. At line n6

F. None of the above

**Answer.** This question investigates how objects become eligible for garbage collection and, at the risk of spoiling the story by giving away the outcome, shows one way that memory leaks are possible in Java.

It turns out that the object created on line n1 never becomes eligible for garbage collection unless a new value is written to the static variable or the class remains loaded in the JVM.

However, class unloading is not a topic of the exam, so from an exam perspective, the object is effectively collectable only if more code is added. If the object never becomes eligible for collection, the memory is simply reclaimed by the operating system after the JVM process exits.

Because the question never addresses when the JVM exits, and that is not an option you can select, the correct option must be F: “None of the above.”

By the way, a while ago, the Oracle exam writers adopted a general policy of avoiding “None of the above” and similar variations as options. However, because this rule has not always been in place, we can’t be completely certain you won’t ever see it. Plus, it suits the learning purpose of this question to use it here.

So, how does an object become eligible for garbage collection? This happens when there are no live references to the object left in the program. What does that mean? Well, when the object of interest is created on line n1, the variable o1 is assigned to point at it. The variable o1 is a reference (a reference is a kind of pointer, but one to which you cannot make arbitrary changes). In other words, o1 is not the data; it’s how to find the data. Because o1 is a variable that the thread can use, it’s a live reference and the object is not eligible for garbage collection because the program can still find the object.
Importantly, anytime you take a copy of the value of \texttt{o1}, you duplicate the instructions on how to find the object (you don’t duplicate the data).

For the sake of a colorful analogy, imagine the object is buried treasure. If a pirate has a treasure map (a reference), he can find the treasure and use it if he wants to. Further, if another pirate makes a copy of that treasure map, either pirate can find the treasure.

Now, if the first pirate’s map sinks with his ship, the second pirate can still find the treasure and use it. But if the second pirate’s ship sinks as well, and there are no further copies of the map, nobody can ever find the treasure. (The assumptions are that the pirates aren’t rescued and they don’t remember the map details in their heads.) This is equivalent to the situation in which an object becomes eligible for garbage collection.

Back to real life: The line right before line \texttt{n2—}\texttt{m.put("o1", o1)—}puts a copy of the reference to the object into a \texttt{Map} (the data structure \texttt{Map}, not the pirate map, although the analogy holds there, too). This means that the \texttt{Map} structure can be used to reach the object.

Next, line \texttt{n2} overwrites the pointer value in \texttt{o1} with the value of \texttt{o2}. This action, in effect, turns the pirate map for finding the object into a pirate map for finding a \textit{different} object. But the original object can still be reached by using the variable \texttt{m}. The variable \texttt{m} lets you find the \texttt{Map} data structure, and the \texttt{Map} data structure still lets you find the original object. So, at this point, the original object is still not lost and is not eligible for collection. That means option A is incorrect.

At line \texttt{n3}, the value of the \texttt{o1} reference variable is changed again, but because it no longer refers to the original object, that doesn’t change the picture. You still can reach the object, so option B is also incorrect.

The next line (between lines \texttt{n3} and \texttt{n4}) makes a copy of the reference value currently in variable \texttt{m}. The copy is placed in the \texttt{List} referred to by the static variable \texttt{1}. That means that you could follow the reference in \texttt{l} to find the \texttt{Map} (which at this point is still also referred to by the variable \texttt{m}), and from the \texttt{Map} you can find the object. Thus, you now have another route for finding that object.

Importantly, the variable \texttt{l} is \texttt{static}, so unlike method local variables \texttt{m}, \texttt{o1}, and \texttt{o2}, which cease to exist when the method \texttt{doIt} returns to its caller, the variable \texttt{l} (that is, the reference variable \texttt{1}, which is distinct from the \texttt{List} to which it refers) will not disappear unless the class \texttt{GCDemo} is unloaded (or the JVM shuts down).

Next, line \texttt{n4} nulls out the direct reference to the \texttt{Map} (that’s the variable \texttt{m}). However, it is still possible to find the \texttt{Map} because
you have the reference to it stored in the List from the previous line. By following the chain from that List to the Map, and then from the Map to the object, you can still reach the object. So, even now, the object is still reachable and thus still not eligible for garbage collection. Therefore, option C is incorrect.

At line n5, the code invokes the System.gc() method, which encourages the garbage collector to invest some time cleaning up. This method has been the subject of much commentary regarding why it might be best avoided, but that isn’t relevant here.

What’s important to remember is that the garbage collector never makes anything eligible for collection; all it ever does is collect things that are already eligible. As you saw, the object of interest wasn’t eligible on line n4, so the call on line n5 changes nothing and, therefore, option D is incorrect.

After the doIt method returns to its caller, the local variables o1, o2, and m cease to exist. Those three pirate maps go down with the sinking ship that is the doIt method. But the static variable l in the GCDemo class still exists and is still accessible. The transitive series of references still leads to the object. So, there is no change at this point.

Line n6 nulls out the reference demo that refers to an instance of the GCDemo class, which happens to render that object eligible for collection. However, the variable l that still lets you reach the object is static, so unless something changes the value of l or the class GCDemo is unloaded, the object remains reachable. From this you can see that option E is also incorrect and, by elimination, option F must be correct.

This kind of behavior is a good candidate for creating a memory leak. Of course, it’s possible this reference chain was kept deliberately, and there might be other code that cleans up the List contents, and the contents of the Map variables that are in that List, at intervals. In that case, everything would be fine, but if this were overlooked, you would likely find that the program consumes ever more memory as it runs, creating a memory leak.

What actions could you take to avoid having this become a memory leak? Several options exist, and the right one depends on the real purpose of the code. Note that there are two potential leaks in the current code. Every call to doIt puts another Map into the List, and in that Map there’s another Object. These instances of Map and Object must be kept under control. Actions that might be involved in keeping memory allocation under control include the following:

- Explicitly removing object references from the Map instance(s)
- Explicitly removing Map references from the List
• Explicitly overwriting the value of the **static** variable `l`, perhaps with `null`.

**Conclusion.** The correct answer is option F.

---

**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**

---

---

---