JAVA SE

Quiz yourself: Correctly apply the static keyword to methods and fields

Test your knowledge of static fields in Java.

by Simon Roberts and Mikalai Zaikin

November 19, 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. 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 of this Java SE 11 quiz is to apply the `static` keyword to methods and fields. Given that this code is compiled:

```
class TestRunner {
  static public int succeeded;
  static public int failed;

  public TestRunner() {
  }

  public TestRunner(int s, int f) {
    succeeded=s; failed=f;
  }

  public String toString() {
      return succeeded + " " + failed;
  }
}

class TestStatistics {
  public static void main(String[] args) {
    TestRunner tr = new TestRunner(1, 99);
    TestRunner.succeeded = 99;
```

```
    System.out.print(new TestRunner());
  }
}
```

And is run as:

```
java TestStatistics
```

What is the output? Choose one.

A. `0 0` The answer is A.

B. `1 99` The answer is B.

C. `99 0` The answer is C.

D. `99 99` The answer is D.

Answer. First, let's acknowledge that this code is a horrible and potentially confusing use of `static` fields, but sometimes good programmers have to handle code written by less-good programmers or by programmers who were in a crushing hurry. That, of course, requires accurate understanding of code even when the design or implementation is questionable. As such, confusing code is legitimate territory for a quiz question.

Static fields have been described in many ways. Sometimes a static field is described as a single value shared by all instances of the class (which doesn't quite capture the situation, since it tends to hide the fact that static fields can exist even in the absence of any instances). Sometimes, it's said that static fields behave as if they are members of the `java.lang.Class` object that defines the class, rather than as members of the instances. That's perhaps more complicated, but it is a better representation, since it addresses the observation that they can exist in the absence of any instances. Note that all instances of any one class share a single `java.lang.Class` object.

However, the essential part of the ideas relevant to this question is that no matter how many objects of the `TestRunner` class are created, each object will see, and potentially modify, the exact same variables, which are called `succeeded` and `failed` .

Let's look at the behavior of the code.

When the class is first loaded, the values of `succeeded` and `failed` will be given default initialization to zero values (the same default that is applied to all numeric fields, whether they are static or instance).

Next, the `main` method creates an instance of `TestRunner`, and the constructor overwrites the values with constructor argument values, which are `1` and `99`.

On the second line of the `main` method's body, there's an explicit assignment to `TestRunner.succeeded` with the value `99`. This writes to the single variable called `succeeded`, so at this point, both `succeeded` and `failed` contain the value `99`.

The final step in the `main` method is to invoke the zero-argument constructor to create a new `TestRunner`. This does not modify the values of anything, so the two static fields both still contain `99`. Right after initialization of the object, the `toString` method is invoked implicitly, and the result of that is printed. Because the values of the static variables are both `99`, the resulting output is the output shown in option D. Therefore option D is correct, and options A, B, and C are incorrect.

Option A would describe the output of the third line of the `main` method if the two fields were instance fields, rather than `static` ones. However, in that situation, the second line of the `main` method would not compile, since you cannot access instance fields using the class name as a prefix.

Option B describes what would happen if the second line of the `main` method (the one that performs the explicit assignment `TestRunner.succeeded = 99`) were not called.

Option C describes what would happen if the first line of the `main` method (the one that says `TestRunner tr = new TestRunner(1, 99);`) were omitted.

There's another point that's worth discussing because it's often described imprecisely and in a way that causes great confusion to newcomers.

The `toString` method—an instance method—accesses the `static` variables. There's no problem here, because the `static` members behave as if they're members of the object (although in that model, they constitute a "single value shared by all instances"). However, it's often said that the opposite is forbidden—that is, that you cannot access an instance variable from a `static` context. That assertion is misleading.

The reality is that when Java comes across an unqualified variable, that is, something like `succeeded`, rather than `xxx.succeeded`, it searches for the target value roughly as follows:

- First, look for a local variable with that name.
- If that search finds nothing, then *if the special reference* `this` *exists*, try treating the variable as if it were prefixed with `this` and see whether that succeeds.
- If that's unsuccessful, try treating the variable as if it were prefixed with the enclosing class name. (These last two steps proceed up the class hierarchy too.)

If there is an explicit prefix, none of this applies, though it might apply to the resolution of that explicit prefix, which might be, in its turn, an unqualified variable!

From this description, you can see that the real issue with `static` methods and instance fields is that `static` methods do not have a `this` reference with which to access instance fields. A `static` method is not prohibited from accessing instance fields, but it cannot do so using the implicit `this` reference, which doesn't exist in the static context.

Conclusion: The correct answer is option D.

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