JAVA SE

Quiz yourself: Implicit and explicit variable context in Java

To properly address any variable, the Java compiler needs to be able to unambiguously determine the context in which it exists.

by Simon Roberts and Mikalai Zaikin

July 16, 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.

Given the following classes

```
public class Test {
    int result = -1;
}

public class CodeTest extends Test {
    public static void main(String[] args) {
                                        // line n1
    }
}
```

Which of the following lines, added independently at line n1, will make the code compile successfully? Choose two.

A. `result = 0;` The answer is A.

B. `this.result = 0;` The answer is B.

C. `super.result = 0;` The answer is C.

D. `new Test().result = 0;` The answer is D.

E. `new CodeTest().result = 0;` The answer is E.

Answer. This question investigates a small part of the rules by which names are resolved, the distinction between static and nonstatic variables, and access to variables from a static context.

This quiz presents a simplified description of some of the rules laid out in *Java Language Specification* section 6 for names and in section 15 for expressions. The discussion here will be nowhere near as complete as in the specification, of course, but we aim to present a perspective that has sufficient scope to answer this question and others like it. The approach presented is sound as far as it goes, but it does have limitations. Undoubtedly some readers will have knowledge way beyond the intended audience of this question, and thus they will likely be able to see exceptions to this simplified description.

Every variable in Java exists in a *context*. (The same is true of methods, although this discussion will mostly focus on variables.) That context can be a method, an object (an instance of a particular class), or a class.

Variables declared inside methods have a *method context* and are often called either a *method local variable* or an *automatic variable*. This quiz question isn't really concerned with variables in method contexts.

Variables declared inside classes—but outside of methods—are in either a class context or an instance context. If such variables carry the `static` modifier, they exist in a *class context*. If such variables do not have the `static` modifier, they exist in an *instance context*. With the instance context, a different variable that has the same name exists in every object of the same type. With the class context, only one variable exists, and that variable is considered to be part of the class.

Given those rules, it's perhaps not a surprise that to address any variable, the context in which it exists must also be identified. This identification can always be done explicitly, but it is often done implicitly.

If you have a class such as this

```
public class Car {
    public static int MAX_SPEED = 105;
    public int speed = 10;
}
```

And you have two objects created from this class

```
Car c1 = new Car();
Car c2 = new Car();
```

Then the variable `speed` is an instance variable and always must be referred to in the context of an instance of the class `Car`. Therefore, provided that variables `c1` and `c2` are in scope, you can write code in terms of `c1.speed` or `c2.speed`, but you cannot refer to `Car.speed`. This is because both `c1` and `c2` refer to contexts that contain a variable called `speed`, whereas the `Car` context does not. It's also worth noting that the contexts referred to by `c1` and `c2` each contain independent

variables that simply share the same basic name. This, of course, allows you to model multiple cars so each has its own speed.

By contrast, the variable `MAX_SPEED` exists in the context of the class `Car`. Therefore, the proper way to refer to it is `Car.MAX_SPEED`. Because the compiler knows that `c1` and `c2` are of the type `Car`, it's also possible to refer to the exact same variable—that is, `Car.MAX_SPEED`—as either `c1.MAX_SPEED` or `c2.MAX_SPEED`. The syntax of those latter two is widely discouraged, but it is likely to show up on an exam precisely because that syntax is potentially confusing, and a competent programmer must not be confused by it.

Notice that `c1.MAX_SPEED` and `c2.MAX_SPEED` look like different variables, but they're not; they are simply allowable aliases for `Car.MAX_SPEED`. That ambiguity is a bad thing when it comes to creating understandable code.

In a few paragraphs, we'll discuss the context called `this`. At the risk of getting ahead of ourselves, know that if `this` exists, it too can be used as a prefix for a static variable. This is the same as using an instance prefix, and it's at least as ugly. If this paragraph confused you, ignore it for now; keep going, and revisit this paragraph after you've read the discussion on `this` coming later.

Up to this point, *explicit contexts* have been discussed. *Implicit contexts* and the meaning of `this` and `super`, which bridge explicit and implicit contexts, also need to be considered.

Let's start with implicit contexts. An implicit context means there's no prefix in front of a variable name. In this case, the compiler must work out where to find the variable from one of three possible contexts.

- First, the compiler checks if there's a method local variable in scope with that name. If one is found, that's what's accessed. Local variables *always* win. If there's no local variable, the compiler looks for class or instance variables, starting at the most specific class definition and working up through the class hierarchy. Only one or the other can possibly exist in one class declaration; otherwise, the code won't compile.
- If this search finds a static variable, that variable is used. Implicitly, the compiler has determined that the context is that of the enclosing class.
- However, if the search finds an instance variable, the context *must* be the value known as `this`. However, `this` exists only in an instance (nonstatic) method. Instance methods must be invoked with a context, and that context becomes the value of `this` inside the method.

Consider the following code fragment:

```
public class Car {
    private int speed;
    public void setSpeed(int s) {
        speed = s;
    }
}
```

And consider the following code:

```
Car c1 = new Car();
c1.setSpeed(55);
```

Inside the code of `setSpeed`, when the call to `c1.setSpeed(55)` is executed, the object referred to by the variable `this` is the same object that was referred to by `c1` at the moment of the call. In other words, the context of the method call is `c1`, and that same context has been embedded inside `this` in that particular method invocation.

A static method cannot access an instance variable using the `this` prefix, either implicitly or explicitly.

Now, the use of an implicit context—the implicit use of `this`—makes sense only if there is a value for `this`. And because that value comes from the instance prefix used for the method invocation, it's perhaps not a surprise that there is no value called `this` in a static method. As a result, a static method cannot access an instance variable using the `this` prefix, either implicitly or explicitly.

As a side note, you could be forgiven for thinking that the ugly syntax mentioned above—using an object prefix to invoke a static method—might create a `this` context in the static method, but it does not. That's another reason the syntax is considered ugly and to be avoided. When a static method is invoked using that syntax, the compiler simply takes the type of the variable prefix and discards the value. Of course, the syntax used to call the method is not known at the time the method is written, nor could you guarantee that all invocations would use the ugly form. Therefore, the call format cannot sensibly affect what is or is not permitted inside the method. The bottom line is that a static method cannot use `this` either implicitly or explicitly.

Another note is that it's common to hear people say, "Static methods cannot access instance variables." That's not accurate; they absolutely can do so, provided the static methods have an explicit object context and that context is not `this`. (The normal rules of access control apply too; so, if the variable in question is in an instance of another class, the variable can't be private, and it might need to be public.)

Moving on, there's a variation on the explicit context `this`, and it is `super`. The `super` keyword means "the object referred to by `this` but viewed with the parent type." Logically then, if `this` doesn't exist, neither does `super`. And if `this` exists (in other words, if you're inside an instance method), `this` and `super` are equal in value but different in type. The type of `super` is the type of the immediate superclass of the type of `this`.

Now that you've reviewed the background, you should be in a good position to evaluate the options and decide on the right answers.

The method that surrounds line n1 is a static method, so you know that no `this` context is available—and neither is a `super` context. This fact is key in answering this question.

The code of option A makes an unqualified reference to a variable called `result`. The compiler will look for a method local variable of that name but fail to find one. Because of that failure, the compiler proceeds to look for a static variable in the context of the class `CodeTest` but

again fails. Because the method is static, no `this` context is available, so the search in the class `CodeTest` will fail. The search then repeats up the parent class hierarchy (in the class `Test` and then in the class `Object`), but, of course, the search fails again. Even though `Test` has a field called `result`, there's still no `this` context from which to access an instance. Therefore, the code fails to compile and option A is incorrect.

Paying close attention to the way that search was just described reveals that it's possible to have a static variable and an instance variable with the same name in the same object, but only if they are declared at different points in the hierarchy. This is a very bad practice because it will cause confusion that will make maintenance harder. Just because syntax is legal doesn't make it good.

In option B, the code tries to make explicit use of the `this` context. But the enclosing method is still static, so no such context exists. The code cannot compile. Therefore, option B is also incorrect.

Option C trades the `this` explicit context for the `super` explicit context, but it was already established that the code is in a static method. Thus, neither `this` nor `super` contexts exist, and the code fails to compile. Therefore, option C is also incorrect.

In option D, the code creates an object of the type `Test`. This is successful, because the class `Test` is accessible and the class has an accessible zero-argument constructor (that's the compiler-generated default constructor, in this case). This newly created object satisfies the need for an explicit object context for the variable `result`, and the code compiles. Therefore, option D is correct.

Option E does almost the same thing as option D, but it creates an instance of `CodeTest` rather than of the base class `Test`. What matters, however, is that the `CodeTest` object forms a context that does contain a variable called `result`. Consequently, the code compiles, and option E is also correct.

By the way, consider the accessibility of the variable `result` in this question. The variable is not specified explicitly and, therefore, it is the default—which means it's accessible from code within the same package. The code shown doesn't indicate a package, so how can you know whether the access succeeds? There are three considerations that might be applied here.

- You could consider that you see all the code and, therefore, both classes are in the default package. In this case, the access would succeed.
- See the Java SE 8 Programmer I exam guidelines and expand the `Review Exam Topics` section, which states the following: "Missing package and import statements: If sample code does not include package or import statements, and the question does not explicitly refer to these missing statements, then assume that all sample code is in the same package, or import statements exist to support them."
- There's no possible way for the first three options (A, B, and C) to compile, but if you assume that the two classes are in the same package, that assumption provides an answer that's believable. Given a question that requires two selections, along with two believable answers and three impossible ones, you have no meaningful choice but to select the two believable ones.

Conclusion. The correct options are D and E.

Related quizzes

- Quiz yourself: Using subclasses and covariant return types
- Quiz yourself: Applying access modifiers to Java classes
- Quiz yourself: Working with abstract classes and default methods in Java

---

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