QUIZ

Quiz Yourself: Wrapper Classes (Intermediate)

Two integers are instantiated with the Integer wrapper class. How do you compare their values correctly?

by Simon Roberts and Mikalai Zaikin

August 26, 2019

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 levels marked “intermediate” and “advanced” refer to the exams, rather than 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 straightforwardly test your knowledge of the ins and outs of the language.

Question (intermediate). The objective is to develop code that uses wrapper classes such as `Boolean`, `Double`, and `Integer`. Given the following code fragment:

```
String one = "1";
Boolean b1 = Boolean.valueOf(one);  // line n1
Integer i1 = new Integer(one);
Integer i2 = 1;
if (b1) {
    System.out.print(i1 == i2);
}
```

What is the result? Choose one.

A. Runtime exception at line n1
B. `true`
C. `false`
D. Code runs with no output

Answer. This question investigates primitive wrapper classes, particularly an odd aspect of the `Boolean` class and its factory. The exact topic of this question is perhaps unlikely on the real exam because it hinges on a piece of rote learning, and the exam tries to avoid such questions. However, this question tests multiple aspects of understanding and knowledge in a single question that, with luck, make it more interesting.

Wrappers provide three main ways to obtain instances. Each wrapper provides a static factory in the method called `valueOf`. Also, if the context is sufficiently explicit, such as an assignment to a variable of the wrapper type from a primitive of matching type, autoboxing will occur. Autoboxing is simply a syntactic shorthand that allows the compiler to write the code to invoke the `valueOf` method and results in cleaner source code. The third approach is to call a constructor, using the `new`

keyword. This third approach actually has been deprecated in Java 9, and one goal of this question is to investigate why it was deprecated.

In Java, anytime a constructor is invoked using the `new` keyword, only two outcomes are possible. Either a new instance of exactly the named type is created and returned, or an exception arises. This is actually a limitation and, today, factory methods are generally preferred because they can have these two effects, but they can also have additional outcomes.

One capability of a factory, which is impossible for a constructor, is to return an *existing* object that matches the request. Given that the `Integer` wrapper is immutable, two objects of this type representing the same number are entirely interchangeable. Because of this, it's a waste of memory to create two objects to represent the same value. Further, such an approach allows comparing such objects using `==` instead of the `equals(Object o)` method. With the `Integer` class, values in the range of -128 to +127 are typically reused in this way.

(As a side note, this behavior is similar in effect to using `String` literals instead of creating a `String` object—such as with `new String("1")`.)

Factory methods have two more advantages. If multiple factories are created, each method can have a different name, which means they could take identical argument type lists if that is appropriate. Doing that would be impossible with constructors, which must form valid overloads of one another.

A third advantage is that a constructor inevitably returns an object of exactly the named type. A factory can return anything that's assignment-compatible with the declared type (including implementations of interfaces, which can be an excellent way of hiding implementation details).

What's most relevant in this question is that when you use `Boolean.valueOf(...)`, you get exactly two constant objects: one `Boolean.TRUE` and one `Boolean.FALSE`. These two are reused as many times as needed without taking up additional memory. This is impossible with calls to `new`.

Now, the factories for most of the wrappers will throw an exception if they are provided with a null argument or with a string that doesn't properly represent the type to be created (for example, if the `Integer.valueOf` factory is called with an argument `"five"` instead of `"5"`). However, the factory for the `java.lang.Boolean` class tests to see whether the argument string exists and contains the value `"true"` (without caring about capitalization). If so, it returns the value `Boolean.TRUE`. If not, it takes no further notice of the argument and returns `Boolean.FALSE`. This means that calling the factory with a null argument, or with the text `"nonsense"`, results in `Boolean.FALSE` and *no* exception is thrown.

Based on this, you can determine that the code in line `n1` does not throw an exception, and it assigns `Boolean.FALSE` to the variable `b1`. Therefore, option A is incorrect.

Now you need to investigate the behavior of the `if` statement and the comparison it contains.

The general rule is that the test expression of an `if` statement must have `boolean` type. It's probably obvious that a `Boolean` object will simply be unboxed to provide that type. If you had any doubt about this, you might wonder if the code would fail to compile. That's not an unreasonable concern, because before the introduction of autoboxing with Java 5, the code would indeed have failed to compile. But even if you had such a concern, there's no option expressing this, so it's safe to assume that the obvious behavior occurs.

In this case, you've established that the object referred to by `b1` represents a `false` value. Because of that, the `if` test fails, and the body of the code is not executed. From that, you can determine that option D is correct.

Although you know it's not executed, in the context of this discussion, it seems worthwhile to consider what would have happened if the argument to the `print` call that's inside the `if` statement were evaluated.

Java provides two forms of equality comparison. One, the `==` operator, is part of the core language. The other, the `equals(Object o)` method, is essentially an API feature. It's available on every object because it's defined in the `java.lang.Object` class, but it doesn't necessarily do anything useful unless a particular class chooses to implement it. It's important to know when to use each of them and to apply them correctly, but this question is concerned only with the `==` operator, so let's just look at that.

The `==` operator compares the values of two expressions. That sounds simple enough except each value can be one of two different types depending on what the basic type of the expression is. This fact is important because the apparent effect of `==` is very different between the two types. By the way, the term "expression" is used here deliberately, and a variable is a simple expression. So if you prefer to think in terms of the values and types of variables, the discussion will still be true; you'll just have a smaller chunk of truth than you might otherwise have had.

The two broad types of an expression are *primitive* (which is one of the eight types `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, and `double`) and what's called a *reference*. A reference is much like a pointer in that it's a value that's used to locate an object elsewhere in memory.

If the expression is a primitive type, the value of the expression actually is the thing you care about. Therefore, if an `int` expression has the value 32, the expression's value actually is a binary representation of 32. Of course it is, you say! Well, yes, but the problem is that if a variable is of a reference type, for example, the type `Integer`, and you use it to refer to an object with the value 32, the value of the variable is *not* 32. Instead, it's some magical number—a reference—that allows the JVM to find the object containing 32. And that means that for reference types (remember: that means *anything* except those eight primitives and, therefore, includes `Integer` expressions) the `==` test tells you not whether the expressions have equivalent meaning, but whether they refer to the *exact same object*. Crucially, if you have two `Integer` objects containing the value 32, but they're different objects, they will have different reference values, and then comparing the expressions that refer to them by using `==` will produce the result `false`.

At this point, it should be clear that if you had this code:

```
Integer v1 = new Integer("1");
Integer v2 = new Integer("1");
System.out.print(v1 == v2);
```

then the output would definitely be `false`. As mentioned earlier, any call to `new` must produce either a new object of exactly the named type or an exception. That means that `v1` and `v2` *must* refer to different objects, and that means the `==` operation must produce `false`.

If instead you had this code:

```
Integer v1 = new Integer("1");
Integer v2 = 1;
System.out.print(v1 == v2);
```

which is closely parallel to the code in the question, using one constructor call and one use of autoboxing would still definitely print `false`. The object created by the constructor will be a unique new object and, therefore, not the one returned by the factory that provides the autoboxed value.

Factories for immutable objects are often coded so that they return the same object every time they're called with the same arguments. The `Integer` class API documentation for the `valueOf(int)` method states the following:

> "This method will always cache values in the range -128 to 127, inclusive, and may cache other values outside of this range."

In other words, the following code:

```
Integer v1 = Integer.valueOf(1);
Integer v2 = Integer.valueOf(1);
System.out.print(v1 == v2);
```

would definitely print `true`.

The guarantee quoted earlier is mentioned only in the documentation for the `valueOf(int)` method and for `valueOf(String)`. However, in practice, both of these methods exhibit the same pooling behavior.

Of course, the question here discusses two `Integer` objects: one created with a constructor and the other using autoboxing (which uses the `Integer.valueOf(int)` method). This means that if the body of the `if` statement had been entered, the output would have been `false`. But you've already established that option D is correct, so options B and C must be incorrect, and this is just an interesting side discussion. We hope it's an interesting one, of course! The correct option is D.

Also in This Issue

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