QUIZ

Quiz Yourself: One-Dimensional Arrays (Advanced)

The subtleties of using a constructor to create an array

by Simon Roberts and Mikalai Zaikin

September 27, 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.

The objective of this question is to declare, instantiate, initialize, and use a one-dimensional array. Given the following code block:

```
int[] ia = new int[2];
ia[1] = 1;
for (int v : ia) System.out.print(v + " ");
String[] sa = new String[2];
sa[0] = "Hi";
for (String v : sa) System.out.print(v + " "); // l
```

What is the output? Choose one.

A. `0 1 Hi null`
B. `null 1 Hi null`
C. `0 1 0 Hi null null`
D. `null 1 null Hi null null`
E. `0 1 Hi`, followed by an exception thrown at line n1

Answer. This question investigates the creation and initialization of arrays. In this example, one-dimensional arrays are used, but multidimensional arrays, which are also part of the exam objectives, are essentially just arrays of arrays. Therefore, although some extra details pertain to them, this discussion will be essentially true for them too.

Arrays can be created in a couple of ways: You can use a constructor form with the keyword `new` or an array literal. This question uses the constructor form, and that form requires that you provide the dimension of the array in square brackets after the type. This method also prevents you from simultaneously specifying values to put into the array.

The literal format, which uses curly braces and is particularly elegant when the array size is small, allows you—and actually *requires* you—to initialize elements explicitly at the same time.

There are two syntactic forms of an array literal. The most general looks like this:

```
new int[] {1, 1}
```

This general form as shown is an expression of type “array of int” and it can be used in any situation that requires an expression of that type. So, for example, it can be used in the actual parameter list of a method call, as follows:

```
doStuffWithIntArray(new int[] {1, 1});
```

Or in a simple assignment, like this:

```
int[] ia;
ia = new int[] {1, 1};
```

And it can be used in other situations.

Notice that in this form (unlike in the constructor form), the square brackets do not contain the array dimension; the array’s size is inferred from the list of initializing values.

A second, perhaps more familiar, form is actually a shorthand and is permitted *only* as the value part of an *initialized variable declaration*. That form can be used only in situations like this:

```
int[] ia = { 1, 1 };
```

Another note is that it’s legitimate to put the square brackets after the variable name in a variable declaration. This form of syntax is less common, because most people like the “type” part of a declaration on the left and the “name” part on the right. However, it was the form used in Java’s precursors, C and C++, so you might come across it. That form looks like this:

```
int ia[];
```

Let’s get back to the question, which doesn’t use these literal forms. Both arrays are created using the constructor form. Therefore, you don’t get the chance to specify the element values as part of the construction process, and any particular values you want to express must be assigned afterwards.

Of course, the explicit assignments in this question do not set all the values of the array, and the crux of this question is what values will be found in the elements of the array to which explicit assignments have not been made.

In fact, this question of “what’s the default value?” applies more generally than just to arrays. Anything allocated on the heap, which includes all objects, whether they are arrays or more obviously object things (the point here is that arrays *are* objects in Java), is subject to a default initialization that’s inseparable from the memory allocation. If you ask for a new object on the heap, the JVM either presents your code with default-initialized memory or an exception. You can never get values that have bypassed this default initialization.

What is the default initialization? Simple: It’s zeros or zero-like values. It’ll be zero for numeric types and `char`, `false` for `boolean` values, and null for all the object reference values.

So, `new int[2]` reserves space for an array with two primitive `int` values. Their indexes (which are also known as subscripts) will be 0 and 1 (Java’s array indexes are always zero-based), and both elements will

have the value 0. Therefore, at this point, you have `ia[0] == 0` and `ia[1] == 0`.

The next line assigns the value 1 to the second element (which has the subscript 1, of course) and leaves the first at its default value of 0.

Next, a simple loop is used to print the values from the `int` array. You've seen that there will be two values: 0 and 1. Therefore, the output from this part of the code will be `0 1` (note that's a `print` method call, not `println`).

The second array is declared and instantiated similarly to the `int` array. It also has one element explicitly initialized, and the other contains the default value. The key difference is that this array has an object type (`String`, specifically), so the default value will be null. Therefore, initially you have `sa[0] == null` and `sa[1] == null`. Next, the element at subscript 0 is assigned to refer to the string `Hi`. During the loop, this code will produce the output `Hi null`.

Based on all this, you can see that option A is correct.

Option B might have been correct if an array of `Integer` types had been created instead of an array of `int`, because `Integer` is an object, and the default value in such an array would again be the null pointer. But that wasn't done, so option B is incorrect.

You might be tempted to select option C or option D if you thought that an array created as `new Blah[x]` has x+1 elements at subscripts 0 through x, rather than having x elements at subscripts 0 through x-1.

Option E would be tempting if you thought that accessing an uninitialized string might cause an exception here. That is not entirely unreasonable. It's perhaps interesting to note that if you evaluate the expression `"String is " + sa[1]`, you would see the output `String is null` ; however, if you evaluate `"String is " + sa[1].toString()`, you would, in fact, get a `NullPointerException`. After all, the value of `sa[1]` is null and trying to use null to refer to an object must fail. However, string concatenation sidesteps that problem and helpfully produces the output `null` instead of throwing the exception.

The correct option is A.

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