Quiz Yourself: Two-Dimensional Arrays (Intermediate)

Using var with two-dimensional arrays requires careful attention.

by Simon Roberts and Mikalai Zaikin

February 27, 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. The “intermediate” and “advanced” designations refer to the exams rather than to 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 to straightforwardly test your knowledge of the ins and outs of the language.

The objective here is to test your ability to use var correctly when you are working with arrays of more than one dimension.

Given the method:

```java
public static void main(String[] args) {
    var[][] arr = {"1", 2},{Integer.valueOf(3)},{4
    System.out.print(arr[0][1]);
}
```

What is the result? Choose one.

A. 2
B. 3
C. Object@1b0375b
D. Compilation fails.

This question investigates declaring and initializing two-dimensional arrays and the use of var in that context.

Java 10 introduced var. Strictly speaking, var is not a keyword or a type. The Java Language Specification for Java 11, in section 3.9 “Keywords,” refers to it as “an identifier with special meaning as the type of a local variable declaration.” We tend to refer to it as “the var pseudo-type” for simplicity.

The var syntax was introduced to allow an alternate form—one that might reduce visual clutter or perhaps provide other benefits—in certain initialized local variable declarations. The general form is that a traditional initialized variable declaration like this:
In this example, it’s not really reasonable to consider the three letters of the type name `int` as “clutter”; however, some variables, particularly those that extensively use nested generics, can be quite verbose. Also, in some situations, the `var` syntax facilitates changing the variable’s type while experimenting with frequently changing prototype code.

We mentioned that the `var` syntax can be used in certain situations, but not all. One of the restrictions is that `var` cannot be used in place of the base type of an array declaration. That’s not the actual wording from the language specification, but that’s probably best, because the relevant section of the specification is somewhat long and refers to relatively abstract syntax specifications. If you’re interested, the discussion is in section 14.4, “Local Variable Declaration Statements.” This particular restriction is commonly misunderstood to mean that `var` cannot be used with array declarations, but that’s not accurate. What it really says is that `var` cannot be used where there are square brackets on the left side of the initializing assignment. This probably needs to be illustrated with examples, so here goes:

This is a regular form initialization that is valid:

```
int ia[] = new int[]{1,2,3}; // VALID
```

However, the following is not a valid use of `var`. Notice that there are square brackets on the left of the assignment:

```
var ia[] = new int[]{1,2,3}; // NOT VALID
```

But the following is permitted. Notice that the square brackets have gone away. The type inferred for `ia` is correctly “array of `int`”:

```
var ia = new int[]{1,2,3}; // VALID
```

Notice that the preceding example includes something else that’s a little unusual: It’s the “new int[]” part. To highlight this, note that the following example is again not valid:

```
var ia = {1,2,3}; // NOT VALID
```

The reason for the failure of this last example, however, is the absence of an explicit base type for the array. This is actually the same reason that you are not permitted to use the simple form of array declaration in a method argument list. For example, given this method declaration:

```
void doStuff(Object[] oa) {}  
```
This invocation is legal:

```java
doStuff(new String[]{"Hello", "Goodbye"}); // VALID
```

While this one is not:

```java
doStuff({"Hello", "Goodbye"}); // NOT VALID
```

The problem is that although it might be clear to you that the array should be "array of String," the compiler cannot infer that with certainty. What if you really wanted to create an array of `Object` that happens to contain `Strings`? The compiler won't guess when there's ambiguity. The same problem arises with the use of `var`—what is the intended base type of the array? Therefore, `var` is not permitted in the situation shown. As a result, the code in the question fails to compile, option D is correct, and options A, B, and C are incorrect.

If you're interested, there are more details and explanation in JEP 286.

It's an interesting side discussion to consider what explicit type you could use for the array's base instead of `var`. You need a single, common Java type that will accept any of the values in the right-side initializer. Taking into account autoboxing, which wraps the primitive values as objects, several types are possible. The most obvious is probably `java.lang.Object`, but `java.io.Serializable` and `java.lang.Comparable` would work too. If you build this, the rest of the code would work:

```java
Object [][] arr = {"1", 2}, {Integer.valueOf(3)}, {4
```

Now consider what would happen if this change were made and the code compiled successfully. The second line of the `main` method prints the value of `arr[0][1]`. In this case, `arr[0]` refers to `{"1", 2}`, and so `arr[0][1]` refers to the `Integer` object wrapping the value 2. So, if the corrected variable declaration were used, the code would print 2 and option A would have been correct.

It's interesting to look at the effect of autoboxing here. If, after having fixed the variable declaration, you ran this code:

```java
System.out.print(arr[0][1].getClass().getCanonicalNa
```

You would see output that shows the primitive 2 has indeed been boxed into an `Integer` object: `java.lang.Integer`.

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.