# Quiz yourself: Java’s scope of variables and instanceof for pattern matching

## When will the instanceof pattern variable be assigned—and what happens if the pattern matching test fails?

[Java 17, which is a long-term support release, is a significant step forward from Java 11. Although Java 17 became generally available in September 2021, the matching version of the certification exam has not been released as of this quiz’s publication date. The exam is in the later stages of development, and we’re excited to begin presenting questions based on the objectives for the new exam. —*Ed.*]

Given the `InstanceTest` class

```
public class InstanceTest {
  public static void main(String[] args) {
    var v = "Java 17";
    doIt(v);
  }
  public static void doIt(Object o) {
    if (!(o instanceof String v)) { // line n1
      throw new IllegalArgumentException("Must be a string");
    }
    if (!(o instanceof Number v)) { // line n2
      System.out.print("Not a number");
    }
  }
}
```

**Which statement is correct?** Choose one.

A. The code runs and throws `IllegalArgumentException` with the description `Must be a string`.

B. The code runs and prints `Not a number`.

C. Compilation fails at line n1.

D. Compilation fails at line n2.

**Answer.** This question explores the pattern matching feature, previewed in Java 14 and released in final form in Java 16. The finished feature is the subject of JEP 394, Pattern matching for `instanceof`.

A long-standing problem in a typical usage of `instanceof` is that two steps, a test and a cast, are required, as follows:

```
if (o instanceof String) {
  String str = (String)o;
  // use str, it has String type
}
```

The pattern matching feature discussed here addresses this problem by creating a simple syntax that performs a test and conditionally initializes a variable of the target type with the reference value of the expression that was tested. A typical usage of this feature looks as follows:

```
if (o instanceof String s) {
  // use s — it is a String already
}
```

The nice aspect of this feature is that by using this syntax, you don’t run the risk of forgetting to perform either of the two parts—the test or the cast—because they are both implicitly, and correctly, bound together in a single syntax.

Notice that this approach consists of two elements.

- There’s a *predicate* part—an `instanceof` test—that is applied to a *target* expression. That’s the variable `o` in the example above: `o instanceof String`.
- There’s a new local variable, also known as the *pattern variable*, which is assigned only if the predicate is true. In the example above, the pattern variable is `s`. Assuming the test passes, `s` is a `String` type reference referring to the same object that the target `o` refers to.

You might wonder what the meaning of `s` is if the test *fails*. And further, what’s the scope of the variable whether the test passes or fails?
To address these questions, Java introduced *flow scoping*, which says that a pattern variable will be in scope *only* when that makes sense because the pattern variable was assigned a value. In other words, if the variable `s` in this example wouldn’t have a meaningful value because the target `o` failed the `instanceof` test, `s` is not in scope.
The following expands the previous example to clarify this point:

```
public void method1(Object o) {
  if (o instanceof String s) {
    // here, s is in scope and correctly refers to
    // object o (which is definitely of type String)
  }
  // here, s does not exist; it’s out of scope
}
```

Sometimes, however, a different test might make this a bit more complex. Suppose you reverse the logic of the test. In that case, *the parts of the code where* `s` *is in scope and where it is not in scope are also reversed*, as follows:

```
public void method2(Object o) {
  if (!(o instanceof String s)) {
    // here, o was not instanceof String
    // and s is out of scope
  } else {
    // here, o was instanceof String,
    // and s is in scope and assigned that String
  }
  // s is also out of scope here
}
```

Let’s continue to explore this by considering a more complex example—which will match the first part of the quiz question.

```
if (!(o instanceof String v)) { // line n1
  throw new IllegalArgumentException("Must be a string");
  // here, v is definitely out of scope
}
// how can you get here?
```

Notice that the negative test ensures the code will throw an exception if presented with a non-`String`. The only way to execute code after the end of the `if` construct is if the code does have a `String`.

The compiler is smart enough to work this out, and in this case, the `String` variable `v` is in scope (and has a valid `String` value) at the point marked `how can you get here?` Because of this, the second test (the one that tests for a `Number`) is attempting to redeclare the variable `v` that is *already* in scope.
Any attempt to redeclare a local variable that's already in scope fails to compile; therefore, line n2 causes a compilation error. Option D is correct, while the other options are incorrect.

```
Error: variable v is already defined in method doIt(Object)
if (!(o instanceof Number v)) { // line n2
                          ^
1 error
```
By the way, option B might have been correct if the code had not definitely thrown an exception in the first test or if the code used a different pattern variable name.

**Conclusion.** The correct answer is option D.

## Dig deeper

- Pattern matching for `instanceof` in Java 14
- JEP 394: Pattern matching for `instanceof`