JAVA SE

Quiz yourself: Working with the Java Path class and the file system

Sometimes a slash is not a slash, as this quiz explains.

by Simon Roberts and Mikalai Zaikin

March 22, 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.

The objective of this quiz is to use the `Path` interface to operate on files and directory paths. Given this code fragment:

```
Path defaultRoot =
    Paths.get(System.getProperty("user.dir"))
Path path = Paths.get("tmp", "john", "..", "d
path = defaultRoot.resolve(path);
System.out.print(path.getName(2)); // line n1
```

Assume that the file system containing the current working directory has an empty directory, `tmp`, in its root.

What is the result? Choose one.

A. `john` The answer is A.

B. `..` The answer is B.

C. `doe` The answer is C.

D. Execution completes without exceptions, The answer is D.

producing output that depends on the host operating system.

E. An exception is thrown at line n1.

The answer is E.

Answer. The `Path` class represents the idea of a path on the file system—that is, an optional sequence of hierarchical directory names, perhaps ending in a filename. A `Path` object itself is not directly linked with the physical file system. Such a connection is created when some action is taken using the `Path`—for example, listing the contents of a directory or creating a file. The reason for avoiding such a hard connection is fairly compelling: If you could use a `Path` only to represent something that already exists, a `Path` could not be used to create a new directory or file.

Given this background, the question revolves around four inquiries. What do the `Paths.get` operations do? What does the `resolve` operation do? Can you extract `getName(2)` from the path after `resolve` has done its work? And if `getName` doesn't throw an exception, what value does it provide in this situation?

You could be forgiven for wondering what the first two lines do in general and, to be fair, some of that code is beyond the scope of the exam. That first line in particular is likely beyond the scope of the real exam, but we left it in to show how you can answer a question even if you don't necessarily have a perfect understanding of everything. Given that none of the options admits the possibility of code other than the last line failing in any way, you can safely assume that these first lines compile and run without crashing. It's also reasonable—and accurate—to assume that this opening code does what it seems to suggest.

Java's APIs give you the tools to work with hierarchical directories without ever making explicit literal references to root directories and path separators.

The first line extracts a `Path` object that represents the root of the file system that contains the user's current working directory. So, for example, if the current working directory is `C:\users\simon\javaprojects\examproj1`, the extracted `Path` object represents `C:\`.

The second line extracts another `Path`—which is likely to be a relative path—representing a path hierarchy that might be represented in a UNIX-like format as either `tmp/john/../doe` or `tmp/doe`.

Which of those two relative paths do you get? From the perspective of `Paths.get`, the `..` is simply an element of the path; it is not treated as anything special in the basic creation process. Therefore, the effective path has four elements: `tmp`, `john`, `..`, and `doe`.

Why do you make the path this way rather than by simply specifying a `String` such as `tmp/john/../doe`? The reason is that Java seeks to allow you to create platform-independent programs. It’s important to be able to describe and manipulate paths on a file system without hardcoding things such as forward slashes or backslashes. Allowing a variable-length argument list of `Strings` and joining them at runtime in whatever way is appropriate for the current execution platform is much more flexible than defining literal strings with separators. You can access the system property `file.separator` to find the local character if you want, but why bother? So, yes, this approach does work, and it looks as if it should. Therefore, option D is incorrect.

As a side note, in literal path strings, forward slashes (UNIX style) work properly on Microsoft Windows–based JVMs, but backslashes (Windows style) *fail* on UNIX-based JVMs. The flexibility on Windows JVMs is possible because a forward slash is not a legal character at the operating system level in Windows paths or filenames. Therefore, if the JVM sees the forward slash, it can safely swap that character for a backslash before handing the path to the Windows system. But if a backslash shows up in a literal path string in UNIX, it’s a legal—if rather odd—character in a UNIX pathname, so simply swapping would change a valid meaning.

This entire question of forward slashes and backslashes is made even more interesting because there are other systems that use different path schemes entirely. On the OpenVMS operating system, local paths have a structure such as `sys$disk:[dir.path.elements]myfile.txt;1`. Clearly, making simple assumptions about slashes would not work with this. The lesson here is that Java’s APIs give you the tools to work with hierarchical directories without ever making explicit literal references to root directories and path separators. Clearly, it’s a good habit to use them, because it will protect your code if it’s ever run on unfamiliar operating systems.

The next question is what does `resolve` do? There are a couple of corner cases, but the most commonly used behavior is the one used here. If the invocation path is not empty and the argument path is not absolute, the effect is to concatenate the two paths. As a result, this takes the relative path (`tmp/john/../doe` in UNIX-like format) and anchors it to the root of the current file system. Again in UNIX-like format, this becomes `/tmp/john/../doe`. Note that `resolve` still does not eliminate the `..` part; that is the job of the method `normalize`.

So, now you need to know whether `getName(2)` is successful and, if it is, what it returns. It turns out that `Path` effectively treats the elements of a path (in this case, `tmp`, `john`, `..`, and `doe`) like a list with a zero-based indexing system. Further, the root part of the path is not included in that list nor in its indexing scheme. Importantly, this exclusion of the root is consistent even if the path were on a Windows system where this example would likely represent `C:\tmp\john\..\doe`. Thus, in a platform-independent way, the indexing starts at zero with `tmp` (and never with `C:\` or something similar). This again shows that option D is incorrect.

Further, you can see that the index 2 should be within the valid range, so option E is incorrect, because no exception is thrown. Further, you will get `..` regardless of the host operating system. This tells you that option B is correct, while options A and C are both incorrect.

Conclusion: The correct answer is option B.

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