JAVA SE

Quiz yourself: The Java FileChannel class

Using a file lock to ensure that only a single instance of an application is running

by Simon Roberts and Mikalai Zaikin

June 21, 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.

Imagine you are writing an application that must be restricted to running a single instance on a single machine. To implement this requirement, you have decided to lock a file when the application starts. If the lock attempt fails, the code generates a message and shuts down the new instance of the application. You have decided to use the `FileLock` class obtained from the `FileChannel` class to implement this, as follows:

```
public class App {
  public static void main(String[] args) thro
    try (FileOutputStream fos = new FileOutpu
         FileChannel channel =  ... // missin
         FileLock lock = channel.tryLock();)
      if (lock == null) {
        System.out.println("Another instance
        System.exit(0);
      }
      System.out.println("Started ...");
      System.console().readLine("Click Enter
    }
  }
}
```

How can you get an instance of the FileChannel to accomplish the requirement? Assume the rest of the code is valid and the file exists on the file system. Choose one.

A. `fos.getChannel();` The answer is A.

B. `new FileChannel(fos);` The answer is B.

C. `FileChannel.open(fos);` The answer is C.

D. `FileSystemProvider.newFileChannel(fos);` The answer is D.

Answer. First, a note about exam style. In general, the exam's questions tend to emphasize understanding and language concepts, rather than simple rote learning. However, there are some rote learning questions, and this is an example of that.

It can be hard to prepare for those questions, since the pool of APIs that are potential candidates for questions is large, and the objectives are not very specific. It seems that the goal for these questions is to get a kind of statistical evidence of breadth of experience. In any case, know that some such questions exist—and spending quality time with the API documentation is *never* time wasted.

The `java.nio.channels.FileChannel` class was added in Java 1.4 as part of the Java NIO API. *NIO* originally meant *new I/O*, though that's not a name that aged well. Today there's a tendency to think of *NIO* as meaning *nonblocking I/O* since this is the package that contains most of the nonblocking I/O facilities other than the `HttpClient`. However, the great majority of the `java.nio` APIs are built around blocking operations.

With file channels you can

- Read from or write to the file at an arbitrary position.
- Truncate the file.
- Append to the file.
- Map the file into memory for faster access, especially for huge files. It's possible that not all the contents of a file will be loaded into memory at once, but the paging effect is hidden from you.
- Transfer data from one channel to another channel, which can be faster than copying I/O streams.
- Create locks on a file (shared or exclusive).

To use a file channel, you must first open it. There are two main approaches for this.

- Use one of the two `public static` overloaded methods from the `FileChannel` class, as follows:

```
FileChannel open(Path path, Set options,
FileChannel open(Path path, OpenOption..
```

- Use the instance method `getChannel()` of the `FileInputStream`, `FileOutputStream`, or `RandomAccessFile` object, as in `public final FileChannel getChannel()`. When the `getChannel()` method is called on an instance of any of these three classes, a `FileChannel` is returned that is connected to the same underlying file. Any change performed using the channel object will be visible through the stream object and vice versa.

Based on the above information, you can see that option A is correct because it demonstrates a valid approach for obtaining a file channel and allows the code to perform properly. Since you're supposed to select only a single option, you should expect the others to be incorrect, but of course, it's informative to check anyway.

Option B is incorrect because the `FileChannel` class is abstract and cannot be instantiated directly. Also, although `FileChannel` does declare a zero-argument constructor, it has `protected` access so it's accessible only in the same package or in a subclass.

Option C is also incorrect. The `FileChannel` class does define two `static open(...)` methods, but they require a `Path` argument (along with other parameters). They do not accept a `FileOutputStream`.

Finally, option D is also incorrect. The class `FileSystemProvider` does indeed have a method `newFileChannel(...)`, which is called by `FileChannel.open(...)`, but it's an instance method, not a `static` method, and it accepts the same list of parameters as the `FileChannel.open(...)` method discussed in option C. Thus, it cannot be invoked with a `FileOutputStream` object.

Here are some additional notes.

First, for the stated question, the code will need an *exclusive lock*, sometimes called a *write lock*, on the file. To do this it's necessary to use a `FileOutputStream` or `RandomAccessFile` opened with `rw` mode. If the channel were obtained from a `FileInputStream`, it's possible to request a shared lock using an overloaded form of either the `lock`or `tryLock` methods, but the form shown in the code would throw an exception, since an exclusive lock is not appropriate for a file opened for reading.

Next, all three classes (`FileOutputStream`, `FileChannel`, and `FileLock`) implement the `java.lang.AutoCloseable` interface. This means those classes can be used directly with a try-with-resource statement.

The `FileChannel.tryLock()` is a nonblocking method and it may be contrasted with the `FileChannel.lock()` method, which will not return until it successfully obtains a lock. The `tryLock` method simply returns null if the file is already locked by another program.

Finally, the implementation of locks is documented to be dependent on the operating system. Some systems might not support the shared or read lock, promoting it to an exclusive or write lock. Also, on some systems, a lock might not actually prevent an operation, but instead it might simply serve as notification to an interested party that the operation should not be performed.

Conclusion. The correct answer is option 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