QUIZ

Quiz Yourself: Threads and Executors (Advanced)

The details of relying on specific operations from ExecutorService

by Simon Roberts and Mikalai Zaikin

August 26, 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.

Question (advanced). The objective is to create worker threads using `Runnable` and `Callable` and to use an `ExecutorService` to concurrently execute tasks. Given the following class:

```
class Logger implements Runnable {
    String msg;
    public Logger(String msg) {
        this.msg = msg;
    }
    public void run() {
        System.out.print(msg);
    }
}
```

And given this code fragment:

```
Stream<Logger> s = Stream.of(
    new Logger("Error "),
    new Logger("Warning "),
    new Logger("Debug "));
ExecutorService es =
    Executors.newCachedThreadPool();
s.sequential().forEach(l -> es.execute(l));
es.shutdown();
es.awaitTermination(10, TimeUnit.SECONDS);
```

Assuming all `import` statements (not shown) are properly configured and the code compiles, which are possible outputs?
Choose two.

A. `Error Debug Warning`
B. `Error Warning Debug`
C. `Error Error Debug`
D. `Error Debug`

Answer. The exam objectives include topics related to the `Executors` class and the `ExecutorService` interface, along with thread pools that implement the `ExecutorService` interface offered by the `Executors` class.

In the first versions of Java, the programmer had the responsibility for creating and managing threads, and because threads are expensive to create and they are limited, kernel-level resources, it was common to create a "thread pool" so a few threads could be used to process a large number of small, independent jobs. The idea of a thread pool is that instead of creating a thread for every little background job and then destroying the threads after each job is completed, a smaller number of threads are created and configured so that packages of work (for example, `Runnable` objects) can be passed to these threads. The first thread that has no work to do takes the job and runs it, and when the job is complete, that thread goes back to look for another job.

Thread pools became part of Java's APIs in Java 5. The `Executor` and `ExecutorService` interfaces are provided as generalizations of thread pools and the interactions they support. Additionally, a number of implementations of `ExecutorService` can be instantiated from a class containing static factory methods. That class is called `Executors`. The `java.util.concurrent` package contains these three types along with many more high-level classes and interfaces aimed at helping a developer address common problems in concurrent programming.

The base interface, `Executor`, can execute tasks that implement `Runnable`. More commonly you use an `ExecutorService`, which is a subinterface of `Executor`. `ExecutorService` adds the ability to process a task that implements the `Callable` interface and to control the shutdown of the thread pool. The `Callable` interface allows a task to produce a result that can be picked up asynchronously by a manager task that is often, but not always, the code that originally submitted the job.

Implementations of `Executor` and `ExecutorService` are not required to use a particular strategy to execute the work submitted to them. Some provide concurrent execution in a fixed-size pool, making new work wait until one of the threads becomes available. Others start more threads when the workload rises and clean them up in times of low demand. Another simply processes the jobs sequentially using a single thread. All of this depends on the particular implementation, and a programmer should be careful to choose something appropriate to the architectural needs of the application. Three factory methods in the `Executors` class produce the behaviors just discussed:

- `newFixedThreadPool`
- `newCachedThreadPool`
- `newSingleThreadExecutor`

While the first two create pools that use multiple worker threads, `newSingleThreadExecutor` creates a service that runs all tasks one after another in a single background thread.

The code in this question uses a cached thread pool. This type of executor service spawns new worker threads as needed and cleans up threads that remain unused for a period of time. However, a cached thread pool has a serious drawback: *There is no maximum limit to the number of threads it might create*. This behavior can lead to high resource consumption and poor performance under high load.

Given that the pool created by the code in the question has multiple threads, you can expect that the jobs submitted to it might well run concurrently. Because of that, regardless of the order they start in, no assumptions can be made about their relative progress. That, in turn, means that the output messages could show up in any order. This tells you that options A and B both are correct.

An `ExecutorService` runs each job that's submitted to it at most one time. There are some circumstances when a job might not be run, or it

could possibly be shut down before completion. However, no job runs more than once. This means that you will definitely not see any duplicated message. Therefore, option C is incorrect.

When you call the `shutdown` method, an `ExecutorService` responds by rejecting any new job requests, but it continues to run until the last job is completed. Because of this, there is no possibility in the given code that you will not see all three messages. Because of this, you know that option D is incorrect.

As a side note, you might wonder whether it's possible to assert that option D is correct on the basis that if the shutdown is not completed in 10 seconds, the code runs off the end of the example; therefore, how can you be sure the messages are printed?

A couple of observations are relevant here. One consideration is that the chances of the jobs shown taking 10 seconds to complete are unreasonably low, and given that two options (A and B) are clearly correct, you can safely reject such an improbable consideration.

Of course, you might still be tempted by option D because an extreme situation in the host might prevent the jobs from completing in 10 seconds. For example, suppose your OS starts installing updates right at the moment the given code is launched. Notice that there's nothing in the given code that shows the virtual machine being shut down forcibly. The threads in this thread pool are nondaemon threads and, therefore, the virtual machine will not shut down until the jobs have been completed. Consequently, the messages will be printed if the program is allowed to run.

The correct options are A and B.

Also in This Issue

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