W

Java 8 Concurrency Tutorial: Threads and Executors

April 07, 2015

Welcome to the first part of my Java 8 Concurrency tutorial. This guide teaches you concurrent programming in Java 8 with easily understood code examples. It's the first part out of a series of tutorials covering the Java Concurrency API. In the next 15 min you learn how to execute code in parallel via threads, tasks and executor services.

- Part 1: Threads and Executors
- Part 2: Synchronization and Locks
- Part 3: Atomic Variables and ConcurrentMap

The Concurrency API was first introduced with the release of Java 5 and then progressively enhanced with every new Java release. The majority of concepts shown in this article also work in older versions of Java. However my code samples focus on Java 8 and make heavy use of lambda expressions and other new features. If you're not yet familiar with lambdas I recommend reading my Java 8 Tutorial first.

Threads and Runnables

All modern operating systems support concurrency both via processes and threads. Processes are instances of programs which typically run independent to each other, e.g. if you start a java program the operating system spawns a new process which runs in parallel to other programs. Inside those processes we can utilize threads to execute code concurrently, so we can make the most out of the available cores of the CPU.

Java supports Threads since JDK 1.0. Before starting a new thread you have to specify the code to be executed by this thread, often called the *task*. This is done by implementing `Runnable` - a functional interface defining a single void no-args method `run()` as demonstrated in the following example:

```
Runnable task = () -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
};

task.run();

Thread thread = new Thread(task);
thread.start();

System.out.println("Done!");
```

Since `Runnable` is a functional interface we can utilize Java 8 lambda expressions to print the current threads name to the console. First we execute the runnable directly on the main thread before starting a new thread.

The result on the console might look like this:

```
Hello main
Hello Thread-0
Done!
```

Or that:

```
Hello main
Done!
Hello Thread-0
```

Due to concurrent execution we cannot predict if the runnable will be invoked before or after printing 'done'. The order is non-deterministic, thus making concurrent programming a complex task in larger applications.

Threads can be put to sleep for a certain duration. This is quite handy to simulate long running tasks in the subsequent code samples of this article:

```
Runnable runnable = () -> {
    try {
        String name = Thread.currentThread().getName();
        System.out.println("Foo " + name);
        TimeUnit.SECONDS.sleep(1);
        System.out.println("Bar " + name);
    }
    catch (InterruptedException e) {
        e.printStackTrace();
    }
};

Thread thread = new Thread(runnable);
thread.start();
```

When you run the above code you'll notice the one second delay between the first and the second print statement. `TimeUnit` is a useful enum for working with units of time. Alternatively you can achieve the same by calling `Thread.sleep(1000)`.

Working with the `Thread` class can be very tedious and error-prone. Due to that reason the **Concurrency API** has been introduced back in 2004 with the release of Java 5. The API is located in package `java.util.concurrent` and contains many useful classes for handling concurrent programming. Since that time the Concurrency API has been enhanced with every new Java release and even Java 8 provides new classes and methods for dealing with concurrency.

Now let's take a deeper look at one of the most important parts of the Concurrency API - the executor services.

Executors

The Concurrency API introduces the concept of an `ExecutorService` as a higher level replacement for working with threads directly. Executors are capable of running asynchronous tasks and typically manage a pool of threads, so we don't have to create new threads manually. All threads of the internal pool will be reused under the hood for revenant tasks, so we can run as many concurrent tasks as we want throughout the life-cycle of our application with a single executor service.

This is how the first thread-example looks like using executors:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
});

// => Hello pool-1-thread-1
```

The class `Executors` provides convenient factory methods for creating different kinds of executor services. In this sample we use an executor with a thread pool of size one.

The result looks similar to the above sample but when running the code you'll notice an important difference: the java process never stops! Executors have to be stopped explicitly - otherwise they keep listening for new tasks.

An `ExecutorService` provides two methods for that purpose: `shutdown()` waits for currently running tasks to finish while `shutdownNow()` interrupts all running tasks and shut the executor down immediately.

This is the preferred way how I typically shutdown executors:

```
try {
    System.out.println("attempt to shutdown executor");
    executor.shutdown();
    executor.awaitTermination(5, TimeUnit.SECONDS);
}
catch (InterruptedException e) {
    System.err.println("tasks interrupted");
}
finally {
    if (!executor.isTerminated()) {
        System.err.println("cancel non-finished tasks");
    }
    executor.shutdownNow();
    System.out.println("shutdown finished");
}
```

The executor shuts down softly by waiting a certain amount of time for termination of currently running tasks. After a maximum of five seconds the executor finally shuts down by interrupting all running tasks.

Callables and Futures

In addition to `Runnable` executors support another kind of task named `Callable`. Callables are functional interfaces just like runnables but instead of being `void` they return a value.

This lambda expression defines a callable returning an integer after sleeping for one second:

```
Callable<Integer> task = () -> {
    try {
        TimeUnit.SECONDS.sleep(1);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
};
```

Callables can be submitted to executor services just like runnables. But what about the callables result? Since `submit()` doesn't wait until the task completes, the executor service cannot return the result of the callable directly. Instead the executor returns a special result of type `Future` which can be used to retrieve the actual result at a later point in time.

```
ExecutorService executor = Executors.newFixedThreadPool(1);
Future<Integer> future = executor.submit(task);

System.out.println("future done? " + future.isDone());

Integer result = future.get();

System.out.println("future done? " + future.isDone());
System.out.print("result: " + result);
```

After submitting the callable to the executor we first check if the future has already been finished execution via `isDone()`. I'm pretty sure this isn't the case since the above callable sleeps for one second before returning the integer.

Calling the method `get()` blocks the current thread and waits until the callable completes before returning the actual result `123`. Now the future is finally done and we see the following result on the console:

```
future done? false
future done? true
result: 123
```

Futures are tightly coupled to the underlying executor service. Keep in mind that every non-terminated future will throw exceptions if you shutdown the executor:

```
executor.shutdownNow();
future.get();
```

You might have noticed that the creation of the executor slightly differs from the previous example. We use `newFixedThreadPool(1)` to create an executor service backed by a thread-pool of size one. This is equivalent to `newSingleThreadExecutor()` but we could later increase the pool size by simply passing a value larger than one.

Timeouts

Any call to `future.get()` will block and wait until the underlying callable has been terminated. In the worst case a callable runs forever - thus making your application unresponsive. You can simply counteract those scenarios by passing a timeout:

```
ExecutorService executor = Executors.newFixedThreadPool(1);

Future<Integer> future = executor.submit(() -> {
    try {
        TimeUnit.SECONDS.sleep(2);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
});

future.get(1, TimeUnit.SECONDS);
```

Executing the above code results in a `TimeoutException`:

```
Exception in thread "main" java.util.concurrent.TimeoutException
    at java.util.concurrent.FutureTask.get(FutureTask.java:205)
```

You might already have guessed why this exception is thrown: We specified a maximum wait time of one second but the callable actually needs two seconds before returning the result.

InvokeAll

Executors support batch submitting of multiple callables at once via `invokeAll()`. This method accepts a collection of callables and returns a list of futures.

```
ExecutorService executor = Executors.newWorkStealingPool();

List<Callable<String>> callables = Arrays.asList(
        () -> "task1",
        () -> "task2",
        () -> "task3");

executor.invokeAll(callables)
    .stream()
    .map(future -> {
        try {
            return future.get();
        }
        catch (Exception e) {
            throw new IllegalStateException(e);
        }
    })
    .forEach(System.out::println);
```

In this example we utilize Java 8 functional streams in order to process all futures returned by the invocation of `invokeAll`. We first map each future to its return value and then print each value to the console. If you're not yet familiar with streams read my Java 8 Stream Tutorial.

InvokeAny

Another way of batch-submitting callables is the method `invokeAny()` which works slightly different to `invokeAll()`. Instead of returning future objects this method blocks until the first callable terminates and returns the result of that callable.

In order to test this behavior we use this helper method to simulate callables with different durations. The method returns a callable that sleeps for a certain amount of time until returning the given result:

```
Callable<String> callable(String result, long sleepSeconds) {
    return () -> {
        TimeUnit.SECONDS.sleep(sleepSeconds);
        return result;
    };
}
```

We use this method to create a bunch of callables with different durations from one to three seconds. Submitting those callables to an executor via `invokeAny()` returns the string result of the fastest callable - in that case task2:

```
ExecutorService executor = Executors.newWorkStealingPool();

List<Callable<String>> callables = Arrays.asList(
    callable("task1", 2),
    callable("task2", 1),
    callable("task3", 3));

String result = executor.invokeAny(callables);
System.out.println(result);

// => task2
```

The above example uses yet another type of executor created via `newWorkStealingPool()`. This factory method is part of Java 8 and returns an executor of type `ForkJoinPool` which works slightly different than normal executors. Instead of using a fixed size thread-pool ForkJoinPools are created for a given parallelism size which per default is the number of available cores of the hosts CPU.

ForkJoinPools exist since Java 7 and will be covered in detail in a later tutorial of this series. Let's finish this tutorial by taking a deeper look at scheduled executors.

Scheduled Executors

We've already learned how to submit and run tasks once on an executor. In order to periodically run common tasks multiple times, we can utilize scheduled thread pools.

A `ScheduledExecutorService` is capable of scheduling tasks to run either periodically or once after a certain amount of time has elapsed.

This code sample schedules a task to run after an initial delay of three seconds has passed:

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

Runnable task = () -> System.out.println("Scheduling: " + System.nanoTime());
ScheduledFuture<?> future = executor.schedule(task, 3, TimeUnit.SECONDS);

TimeUnit.MILLISECONDS.sleep(1337);

long remainingDelay = future.getDelay(TimeUnit.MILLISECONDS);
System.out.printf("Remaining Delay: %sms", remainingDelay);
```

Scheduling a task produces a specialized future of type `ScheduledFuture` which - in addition to `Future` - provides the method `getDelay()` to retrieve the remaining delay. After this delay has elapsed the task will be executed concurrently.

In order to schedule tasks to be executed periodically, executors provide the two methods `scheduleAtFixedRate()` and `scheduleWithFixedDelay()`. The first method is capable of executing tasks with a fixed time rate, e.g. once every second as demonstrated in this example:

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

Runnable task = () -> System.out.println("Scheduling: " + System.nanoTime());

int initialDelay = 0;
int period = 1;
executor.scheduleAtFixedRate(task, initialDelay, period, TimeUnit.SECONDS);
```

Additionally this method accepts an initial delay which describes the leading wait time before the task will be executed for the first time.

Please keep in mind that `scheduleAtFixedRate()` doesn't take into account the actual duration of the task. So if you specify a period of one second but the task needs 2 seconds to be executed then the thread pool will working to capacity very soon.

In that case you should consider using `scheduleWithFixedDelay()` instead. This method works just like the counterpart described above. The difference is that the wait time period applies between the end of a task and the start of the next task. For example:

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

Runnable task = () -> {
    try {
        TimeUnit.SECONDS.sleep(2);
        System.out.println("Scheduling: " + System.nanoTime());
    }
    catch (InterruptedException e) {
        System.err.println("task interrupted");
    }
};

executor.scheduleWithFixedDelay(task, 0, 1, TimeUnit.SECONDS);
```

This example schedules a task with a fixed delay of one second between the end of an execution and the start of the next execution. The initial delay is zero and the tasks duration is two seconds. So we end up with an execution interval of 0s, 3s, 6s, 9s and so on. As you can see `scheduleWithFixedDelay()` is handy if you cannot predict the duration of the scheduled tasks.

This was the first part out of a series of concurrency tutorials. I recommend practicing the shown code samples by your own. You find all code samples from this article on GitHub, so feel free to fork the repo and give me star.

I hope you've enjoyed this article. If you have any further questions send me your feedback in the comments below or via Twitter.

- Part 1: Threads and Executors

- Part 2: Synchronization and Locks
- Part 3: Atomic Variables and ConcurrentMap

Follow @winterbe Follow @winterbe_ Tweet

Benjamin is Software Engineer, Full Stack Developer at Pondus, an excited runner and table foosball player. Get in touch on Twitter and GitHub.

Do you know Sequency?

Read More

Recent | All Posts | Java | Tutorials

Integrating React.js into Existing jQuery Web Applications

Java 8 Concurrency Tutorial: Atomic Variables and ConcurrentMap

Java 8 Concurrency Tutorial: Synchronization and Locks

Java 8 API by Example: Strings, Numbers, Math and Files