



Concurrency Utilities and Executors

The previous four chapters focused on Java's low-level support for threads. This chapter switches that focus to Java's high-level thread support, which is known as the concurrency utilities. Think of the concurrency utilities as being analogous to writing applications in a high-level language and its low-level thread support as being analogous to writing applications in assembly language. After briefly introducing you to these utilities, I take you on a tour of executors. The next three chapters will cover other subsets of the various concurrency utilities.

Introducing the Concurrency Utilities

Java's low-level threads support lets you create multithreaded applications that offer better performance and responsiveness over their single-threaded counterparts. However, there are problems:

- Low-level concurrency primitives such as `synchronized` and `wait()/notify()` are often hard to use correctly. Incorrect use of these primitives can result in race conditions, thread starvation, deadlock, and other hazards, which can be hard to detect and debug.
- Too much reliance on the `synchronized` primitive can lead to performance issues, which affect an application's *scalability*. This is a significant problem for highly-threaded applications such as web servers.
- Developers often need higher-level constructs such as thread pools and semaphores. Because these constructs aren't included with Java's low-level thread support, developers have been forced to build their own, which is a time-consuming and error-prone activity.

To address these problems, Java 5 introduced the *concurrency utilities*, a powerful and extensible framework of high-performance threading utilities such as thread pools and blocking queues. This framework consists of various types in the following packages:

- `java.util.concurrent`: Utility types that are often used in concurrent programming, for example, executors.
- `java.util.concurrent.atomic`: Utility classes that support lock-free thread-safe programming on single variables.
- `java.util.concurrent.locks`: Utility types that lock and wait on *conditions* (objects that let threads suspend execution [wait] until notified by other threads that some boolean state may now be true). Locking and waiting via these types is more performant and flexible than doing so via Java's monitor-based synchronization and wait/notification mechanisms.

This framework also introduces a long `nanoTime()` method to the `java.lang.System` class, which lets you access a nanosecond-granularity time source for making relative time measurements.

The concurrency utilities can be classified as executors, synchronizers, a locking framework, and more. I explore executors in the next section and these other categories in subsequent chapters.

Exploring Executors

The Threads API lets you execute runnable tasks via expressions such as `new java.lang.Thread(new RunnableTask()).start();`. These expressions tightly couple task submission with the task's execution mechanics (run on the current thread, a new thread, or a thread arbitrarily chosen from a *pool* [group] of threads).

■ **Note** A *task* is an object whose class implements the `java.lang.Runnable` interface (a runnable task) or the `java.util.concurrent.Callable` interface (a callable task). I'll say more about `Callable` later in this chapter.

The concurrency utilities include executors as a high-level alternative to low-level thread expressions for executing runnable tasks. An *executor* is an object whose class directly or indirectly implements the `java.util.concurrent.Executor` interface, which decouples task submission from task-execution mechanics.

■ **Note** The Executor Framework's use of interfaces to decouple task submission from task-execution is analogous to the Collections Framework's use of core interfaces to decouple lists, sets, queues, and maps from their implementations. Decoupling results in flexible code that's easier to maintain.

Executor declares a solitary `void execute(Runnable runnable)` method that executes the runnable task named `runnable` at some point in the future. `execute()` throws `java.lang.NullPointerException` when `runnable` is null and `java.util.concurrent.RejectedExecutionException` when it cannot execute `runnable`.

■ **Note** `RejectedExecutionException` can be thrown when an executor is shutting down and doesn't want to accept new tasks. Also, this exception can be thrown when the executor doesn't have enough room to store the task (perhaps the executor uses a bounded blocking queue to store tasks and the queue is full—I discuss blocking queues in Chapter 8).

The following example presents the Executor equivalent of the aforementioned `new Thread(new RunnableTask()).start();` expression:

```
Executor executor = ...; // ... represents some executor creation
executor.execute(new RunnableTask());
```

Although Executor is easy to use, this interface is limited in various ways:

- Executor focuses exclusively on `Runnable`. Because `Runnable`'s `run()` method doesn't return a value, there's no easy way for a runnable task to return a value to its caller.
- Executor doesn't provide a way to track the progress of runnable tasks that are executing, cancel an executing runnable task, or determine when the runnable task finishes execution.
- Executor cannot execute a collection of runnable tasks.
- Executor doesn't provide a way for an application to shut down an executor (much less properly shut down an executor).

These limitations are addressed by the `java.util.concurrent.ExecutorService` interface, which extends `Executor` and whose implementation is typically a thread pool. Table 5-1 describes `ExecutorService`'s methods.

Table 5-1. *ExecutorService's Methods*

Method	Description
<code>boolean awaitTermination(long timeout, TimeUnit unit)</code>	Block (wait) until all tasks have finished after a shutdown request, the <code>timeout</code> (measured in unit time units) expires, or the current thread is interrupted, whichever happens first. Return <code>true</code> when this executor has terminated and <code>false</code> when the <code>timeout</code> elapses before termination. This method throws <code>java.lang.InterruptedException</code> when interrupted.
<code><T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)</code>	Execute each callable task in the <code>tasks</code> collection and return a <code>java.util.List</code> of <code>java.util.concurrent.Future</code> instances (discussed later in this chapter) that hold task statuses and results when all tasks complete—a task completes through normal termination or by throwing an exception. The <code>List</code> of <code>Futures</code> is in the same sequential order as the sequence of tasks returned by <code>tasks</code> ' iterator. This method throws <code>InterruptedException</code> when it's interrupted while waiting, in which case unfinished tasks are canceled; <code>NullPointerException</code> when <code>tasks</code> or any of its elements is <code>null</code> ; and <code>RejectedExecutionException</code> when any one of <code>tasks</code> ' tasks cannot be scheduled for execution.
<code><T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)</code>	Execute each callable task in the <code>tasks</code> collection and return a <code>List</code> of <code>Future</code> instances that hold task statuses and results when all tasks complete—a task completes through normal termination or by throwing an exception—or the <code>timeout</code> (measured in unit time units) expires. Tasks that are not completed at expiry are canceled. The <code>List</code> of <code>Futures</code> is in the same sequential order as the sequence of tasks returned by <code>tasks</code> ' iterator. This method throws <code>InterruptedException</code> when it's interrupted while waiting (unfinished tasks are canceled). It also throws <code>NullPointerException</code> when <code>tasks</code> , any of its elements, or <code>unit</code> is <code>null</code> ; and throws <code>RejectedExecutionException</code> when any one of <code>tasks</code> ' tasks cannot be scheduled for execution.

(continued)

Table 5-1. (continued)

Method	Description
<code><T> T invokeAny(Collection<? extends Callable<T>> tasks)</code>	Execute the given tasks, returning the result of an arbitrary task that's completed successfully (in other words, without throwing an exception), if any does. On normal or exceptional return, tasks that haven't completed are canceled. This method throws <code>InterruptedException</code> when it's interrupted while waiting, <code>NullPointerException</code> when tasks or any of its elements is null, <code>java.lang.IllegalArgumentException</code> when tasks is empty, <code>java.util.concurrent.ExecutionException</code> when no task completes successfully, and <code>RejectedExecutionException</code> when none of the tasks can be scheduled for execution.
<code><T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)</code>	Execute the given tasks, returning the result of an arbitrary task that's completed successfully (no exception was thrown), if any does before the timeout (measured in unit time units) expires—tasks that are not completed at expiry are canceled. On normal or exceptional return, tasks that have not completed are canceled. This method throws <code>InterruptedException</code> when it's interrupted while waiting; <code>NullPointerException</code> when tasks, any of its elements, or unit is null; <code>IllegalArgumentException</code> when tasks is empty; <code>java.util.concurrent.TimeoutException</code> when the timeout elapses before any task successfully completes; <code>ExecutionException</code> when no task completes successfully; and <code>RejectedExecutionException</code> when none of the tasks can be scheduled for execution.
<code>boolean isShutdown()</code>	Return true when this executor has been shut down; otherwise, return false.
<code>boolean isTerminated()</code>	Return true when all tasks have completed following shutdown; otherwise, return false. This method will never return true prior to <code>shutdown()</code> or <code>shutdownNow()</code> being called.
<code>void shutdown()</code>	Initiate an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted. Calling this method has no effect after the executor has shut down. This method doesn't wait for previously submitted tasks to complete execution. Use <code>awaitTermination()</code> when waiting is necessary.

(continued)

Table 5-1. (continued)

Method	Description
<code>List<Runnable> shutdownNow()</code>	Attempt to stop all actively executing tasks, halt the processing of waiting tasks, and return a list of the tasks that were awaiting execution. There are no guarantees beyond best-effort attempts to stop processing actively executing tasks. For example, typical implementations will cancel via <code>Thread.interrupt()</code> , so any task that fails to respond to interrupts may never terminate.
<code><T> Future<T> submit(Callable<T> task)</code>	Submit a callable task for execution and return a <code>Future</code> instance representing task's pending results. The <code>Future</code> instance's <code>get()</code> method returns task's result on successful completion. This method throws <code>RejectedExecutionException</code> when task cannot be scheduled for execution and <code>NullPointerException</code> when task is null. If you would like to immediately block while waiting for a task to complete, you can use constructions of the form <code>result = exec.submit(aCallable).get();</code> .
<code>Future<?> submit(Runnable task)</code>	Submit a runnable task for execution and return a <code>Future</code> instance representing task's pending results. The <code>Future</code> instance's <code>get()</code> method returns task's result on successful completion. This method throws <code>RejectedExecutionException</code> when task cannot be scheduled for execution and <code>NullPointerException</code> when task is null.
<code><T> Future<T> submit(Runnable task, T result)</code>	Submit a runnable task for execution and return a <code>Future</code> instance whose <code>get()</code> method returns <code>result</code> 's value on successful completion. This method throws <code>RejectedExecutionException</code> when task cannot be scheduled for execution and <code>NullPointerException</code> when task is null.

Table 5-1 refers to `java.util.concurrent.TimeUnit`, an enum that represents time durations at given units of granularity: `DAYS`, `HOURS`, `MICROSECONDS`, `MILLISECONDS`, `MINUTES`, `NANOSECONDS`, and `SECONDS`. Furthermore, `TimeUnit` declares methods for converting across units (such as `long toHours(long duration)`), and for performing timing and delay operations (such as `void sleep(long timeout)`) in these units.

Table 5-1 also refers to *callable tasks*. Unlike `Runnable`, whose `void run()` method cannot return a value and throw checked exceptions, `Callable<V>`'s `V call()` method returns a value and can throw checked exceptions because it's declared with a `throws Exception` clause.

Finally, Table 5-1 refers to the `Future` interface, which represents the result of an asynchronous computation. The result is known as a *future* because it typically will not be available until some moment in the future. `Future`, whose generic type is `Future<V>`, provides methods for canceling a task, for returning a task's value, and for determining whether or not the task has finished. Table 5-2 describes `Future`'s methods.

Table 5-2. *Future's Methods*

Method	Description
<code>boolean cancel(boolean mayInterruptIfRunning)</code>	Attempt to cancel execution of this task and return <code>true</code> when the task is canceled; otherwise, return <code>false</code> (the task may have completed normally before <code>cancel()</code> was called). Cancellation fails when the task is done, canceled, or couldn't be canceled for another reason. If successful and this task hadn't yet started, the task should never run. If the task has started, <code>mayInterruptIfRunning</code> determines whether (<code>true</code>) or not (<code>false</code>) the thread running this task should be interrupted in an attempt to stop the task. After returning, subsequent calls to <code>isDone()</code> always return <code>true</code> ; <code>isCancelled()</code> always return <code>true</code> when <code>cancel()</code> returns <code>true</code> .
<code>V get()</code>	Wait if necessary for the task to complete and then return the result. This method throws <code>java.util.concurrent.CancellationException</code> when the task was canceled prior to this method being called, <code>ExecutionException</code> when the task threw an exception, and <code>InterruptedException</code> when the current thread was interrupted while waiting.
<code>V get(long timeout, TimeUnit unit)</code>	Wait at most <code>timeout</code> units (as specified by <code>unit</code>) for the task to complete and then return the result (if available). This method throws <code>CancellationException</code> when the task was canceled prior to this method being called, <code>ExecutionException</code> when the task threw an exception, <code>InterruptedException</code> when the current thread was interrupted while waiting, and <code>TimeoutException</code> when this method's <code>timeout</code> value expires (the wait times out).
<code>boolean isCancelled()</code>	Return <code>true</code> when this task was canceled before it completed normally; otherwise, return <code>false</code> .
<code>boolean isDone()</code>	Return <code>true</code> when this task completed; otherwise, return <code>false</code> . Completion may be due to normal termination, an exception, or cancellation—this method returns <code>true</code> in all of these cases.

Suppose you intend to write an application whose graphical user interface lets the user enter a word. After the user enters the word, the application presents this word to several online dictionaries and obtains each dictionary's entry. These entries are subsequently displayed to the user.

Because online access can be slow, and because the user interface should remain responsive (perhaps the user might want to end the application), you offload the “obtain word entries” task to an executor that runs this task on a separate thread. The following example uses `ExecutorService`, `Callable`, and `Future` to accomplish this objective:

```
ExecutorService executor = ...; // ... represents some executor creation
Future<String[]> taskFuture =
    executor.submit(new Callable<String[]>()
    {
        @Override
        public String[] call()
        {
            String[] entries = ...;
            // Access online dictionaries
            // with search word and populate
            // entries with their resulting
            // entries.
            return entries;
        }
    });
// Do stuff.
String entries = taskFuture.get();
```

After obtaining an executor in some manner (you will learn how shortly), the example's thread submits a callable task to the executor. The `submit()` method immediately returns with a reference to a `Future` object for controlling task execution and accessing results. The thread ultimately calls this object's `get()` method to get these results.

■ **Note** The `java.util.concurrent.ScheduledExecutorService` interface extends `ExecutorService` and describes an executor that lets you schedule tasks to run once or to execute periodically after a given delay.

Although you could create your own `Executor`, `ExecutorService`, and `ScheduledExecutorService` implementations (such as class `DirectExecutor` implements `Executor` { `@Override public void execute(Runnable r) { r.run(); }` }—run executor directly on the calling thread), there's a simpler alternative: `java.util.concurrent.Executors`.

■ **Tip** If you intend to create your own `ExecutorService` implementations, you will find it helpful to work with the `java.util.concurrent.AbstractExecutorService` and `java.util.concurrent.FutureTask` classes.

The Executors utility class declares several class methods that return instances of various `ExecutorService` and `ScheduledExecutorService` implementations (and other kinds of instances). This class's static methods accomplish the following tasks:

- Create and return an `ExecutorService` instance that's configured with commonly used configuration settings.
- Create and return a `ScheduledExecutorService` instance that's configured with commonly used configuration settings.
- Create and return a “wrapped” `ExecutorService` or `ScheduledExecutorService` instance that disables reconfiguration of the executor service by making implementation-specific methods inaccessible.
- Create and return a `java.util.concurrent.ThreadFactory` instance (that is, an instance of a class that implements the `ThreadFactory` interface) for creating new `Thread` objects.
- Create and return a `Callable` instance out of other closure-like forms so that it can be used in execution methods that require `Callable` arguments (such as `ExecutorService`'s `submit(Callable)` method). Wikipedia's “Closure (computer science)” entry at [http://en.wikipedia.org/wiki/Closure_\(computer_science\)](http://en.wikipedia.org/wiki/Closure_(computer_science)) introduces the topic of closures.

For example, static `ExecutorService newFixedThreadPool(int nThreads)` creates a thread pool that reuses a fixed number of threads operating off of a shared unbounded queue. At most, `nThreads` threads are actively processing tasks. If additional tasks are submitted when all threads are active, they wait in the queue for an available thread.

If any thread terminates because of a failure during execution before the executor shuts down, a new thread will take its place when needed to execute subsequent tasks. The threads in the pool will exist until the executor is explicitly shut down. This method throws `IllegalArgumentException` when you pass zero or a negative value to `nThreads`.

■ **Note** Thread pools are used to eliminate the overhead from having to create a new thread for each submitted task. Thread creation isn't cheap, and having to create many threads could severely impact an application's performance.

You would commonly use executors, runnables, callables, and futures in file and network input/output contexts. Performing a lengthy calculation offers another scenario where you could use these types. For example, Listing 5-1 uses an executor, a callable, and a future in a calculation context of Euler's number e (2.71828...).

Listing 5-1. Calculating Euler's Number e

```
import java.math.BigDecimal;
import java.math.MathContext;
import java.math.RoundingMode;

import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class CalculateE
{
    final static int LASTITER = 17;

    public static void main(String[] args)
    {
        ExecutorService executor = Executors.newFixedThreadPool(1);
        Callable<BigDecimal> callable;
        callable = new Callable<BigDecimal>()
        {
            @Override
            public BigDecimal call()
            {
                MathContext mc =
                    new MathContext(100, RoundingMode.HALF_UP);
                BigDecimal result = BigDecimal.ZERO;
                for (int i = 0; i <= LASTITER; i++)
                {
                    BigDecimal factorial =
                        factorial(new BigDecimal(i));
                    BigDecimal res = BigDecimal.ONE.divide(factorial,
                                                                mc);
                    result = result.add(res);
                }
                return result;
            }
        }
    }
}
```

```

        public BigDecimal factorial(BigDecimal n)
        {
            if (n.equals(BigDecimal.ZERO))
                return BigDecimal.ONE;
            else
                return n.multiply(factorial(n.
                    subtract(BigDecimal.ONE)));
        }
    };
    Future<BigDecimal> taskFuture = executor.submit(callable);
    try
    {
        while (!taskFuture.isDone())
            System.out.println("waiting");
        System.out.println(taskFuture.get());
    }
    catch (ExecutionException ee)
    {
        System.err.println("task threw an exception");
        System.err.println(ee);
    }
    catch (InterruptedException ie)
    {
        System.err.println("interrupted while waiting");
    }
    executor.shutdownNow();
}
}

```

The default main thread that executes `main()` first obtains an executor by calling `Executors.newFixedThreadPool()` method. It then instantiates an anonymous class that implements the `Callable` interface and submits this task to the executor, receiving a `Future` instance in response.

After submitting a task, a thread typically does some other work until it requires the task's result. I simulate this work by having the main thread repeatedly output a waiting message until the `Future` instance's `isDone()` method returns `true`. (In a realistic application, I would avoid this looping.) At this point, the main thread calls the instance's `get()` method to obtain the result, which is then output. The main thread then shuts down the executor.

■ **Caution** It's important to shut down an executor after it completes; otherwise, the application might not end. The previous executor accomplishes this task by calling `shutdownNow()`. (You could also use the `shutdown()` method.)

The callable's `call()` method calculates e by evaluating the mathematical power series $e = 1 / 0! + 1 / 1! + 1 / 2! + \dots$. This series can be evaluated by summing $1 / n!$, where n ranges from 0 to infinity (and $!$ stands for factorial).

`call()` first instantiates `java.math.MathContext` to encapsulate a *precision* (number of digits) and a rounding mode. I chose 100 as an upper limit on e 's precision, and I also chose `HALF_UP` as the rounding mode.

■ **Tip** Increase the precision as well as the value of `LASTITER` to converge the series to a lengthier and more accurate approximation of e .

`call()` next initializes a `java.math.BigDecimal` local variable named `result` to `BigDecimal.ZERO`. It then enters a loop that calculates a factorial, divides `BigDecimal.ONE` by the factorial, and adds the division result to `result`.

The `divide()` method takes the `MathContext` instance as its second argument to provide rounding information. (If I specified 0 as the precision for the math context and a *nonterminating decimal expansion* [the quotient result of the division cannot be represented exactly—0.3333333..., for example] occurred, `java.lang.ArithmeticException` would be thrown to alert the caller to the fact that the quotient cannot be represented exactly. The executor would rethrow this exception as `ExecutionException`.)

Compile Listing 5-1 as follows:

```
javac CalculateE.java
```

Run the resulting application as follows:

```
java CalculateE
```

You should observe output that's similar to the following (you'll probably observe more waiting messages):

```
waiting
waiting
waiting
waiting
waiting
2.71828182845904507051604779584860506117897963525103269890073500406522504250
4843314055887974344245741730039454062711
```

EXERCISES

The following exercises are designed to test your understanding of Chapter 5's content:

1. What are the concurrency utilities?
2. Identify the packages in which the concurrency utilities types are stored.
3. Define task.
4. Define executor.
5. Identify the `Executor` interface's limitations.
6. How are `Executor`'s limitations overcome?
7. What differences exist between `Runnable`'s `run()` method and `Callable`'s `call()` method?
8. True or false: You can throw checked and unchecked exceptions from `Runnable`'s `run()` method but can only throw unchecked exceptions from `Callable`'s `call()` method.
9. Define future.
10. Describe the `Executors` class's `newFixedThreadPool()` method.
11. Refactor the following `CountingThreads` application to work with `Executors` and `ExecutorService`:

```
public class CountingThreads
{
    public static void main(String[] args)
    {
        Runnable r = new Runnable()
        {
            @Override
            public void run()
            {
                String name = Thread.currentThread().
                    getName();
                int count = 0;
                while (true)
                    System.out.println(name + ": " +
                        count++);
            }
        };
    }
}
```

```

        Thread thdA = new Thread(r);
        Thread thdB = new Thread(r);
        thdA.start();
        thdB.start();
    }
}

```

12. When you execute the previous exercise's `CountingThreads` application, you'll observe output that identifies the threads via names such as `pool-1-thread-1`. Modify `CountingThreads` so that you observe names A and B. Hint: You'll need to use `ThreadFactory`.
-

Summary

Java's low-level thread capabilities let you create multithreaded applications that offer better performance and responsiveness over their single-threaded counterparts. However, performance issues that affect an application's scalability and other problems resulted in Java 5's introduction of the concurrency utilities.

The concurrency utilities organize various types into three packages: `java.util.concurrent`, `java.util.concurrent.atomic`, and `java.util.concurrent.locks`. Basic types for executors, thread pools, concurrent hashmaps, and other high-level concurrency constructs are stored in `java.util.concurrent`; classes that support lock-free, thread-safe programming on single variables are stored in `java.util.concurrent.atomic`; and types for locking and waiting on conditions are stored in `java.util.concurrent.locks`.

An executor decouples task submission from task-execution mechanics and is described by the `Executor`, `ExecutorService`, and `ScheduledExecutorService` interfaces. You obtain an executor by calling one of the utility methods in the `Executors` class. Executors are associated with callables and futures.

Chapter 6 presents synchronizers.