

Item 68: Prefer executors and tasks to threads

The first edition of this book contained code for a simple *work queue* [Bloch01, Item 50]. This class allowed clients to enqueue work items for asynchronous processing by a background thread. When the work queue was no longer needed, the client could invoke a method to ask the background thread to terminate itself gracefully after completing any work that was already on the queue. The implementation was little more than a toy, but even so, it required a full page of subtle, delicate code, of the sort that is prone to safety and liveness failures if you don't get it just right. Luckily, there is no reason to write this sort of code anymore.

In release 1.5, `java.util.concurrent` was added to the Java platform. This package contains an *Executor Framework*, which is a flexible interface-based task execution facility. Creating a work queue that is better in every way than the one in the first edition of this book requires but a single line of code:

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

Here is how to submit a runnable for execution:

```
executor.execute(runnable);
```

And here is how to tell the executor to terminate gracefully (if you fail to do this, it is likely that your VM will not exit):

```
executor.shutdown();
```

You can do *many* more things with an executor service. For example, you can wait for a particular task to complete (as in the “background thread SetObserver” in Item 67, page 267), you can wait for any or all of a collection of tasks to complete (using the `invokeAny` or `invokeAll` methods), you can wait for the executor service's graceful termination to complete (using the `awaitTermination` method), you can retrieve the results of tasks one by one as they complete (using an `ExecutorCompletionService`), and so on.

If you want more than one thread to process requests from the queue, simply call a different static factory that creates a different kind of executor service called a *thread pool*. You can create a thread pool with a fixed or variable number of threads. The `java.util.concurrent.Executors` class contains static factories that provide most of the executors you'll ever need. If, however, you want some-

thing out of the ordinary, you can use the `ThreadPoolExecutor` class directly. This class lets you control nearly every aspect of a thread pool's operation.

Choosing the executor service for a particular application can be tricky. If you're writing a small program, or a lightly loaded server, using `Executors.newCachedThreadPool` is generally a good choice, as it demands no configuration and generally "does the right thing." But a cached thread pool is not a good choice for a heavily loaded production server! In a cached thread pool, submitted tasks are not queued but immediately handed off to a thread for execution. If no threads are available, a new one is created. If a server is so heavily loaded that all of its CPUs are fully utilized, and more tasks arrive, more threads will be created, which will only make matters worse. Therefore, in a heavily loaded production server, you are much better off using `Executors.newFixedThreadPool`, which gives you a pool with a fixed number of threads, or using the `ThreadPoolExecutor` class directly, for maximum control.

Not only should you refrain from writing your own work queues, but you should generally refrain from working directly with threads. The key abstraction is no longer `Thread`, which served as both the unit of work and the mechanism for executing it. Now the unit of work and mechanism are separate. The key abstraction is the unit of work, which is called a *task*. There are two kinds of tasks: `Runnable` and its close cousin, `Callable` (which is like `Runnable`, except that it returns a value). The general mechanism for executing tasks is the *executor service*. If you think in terms of tasks and let an executor service execute them for you, you gain great flexibility in terms of selecting appropriate execution policies. In essence, the Executor Framework does for execution what the Collections Framework did for aggregation.

The Executor Framework also has a replacement for `java.util.Timer`, which is `ScheduledThreadPoolExecutor`. While it is easier to use a timer, a scheduled thread pool executor is much more flexible. A timer uses only a single thread for task execution, which can hurt timing accuracy in the presence of long-running tasks. If a timer's sole thread throws an uncaught exception, the timer ceases to operate. A scheduled thread pool executor supports multiple threads and recovers gracefully from tasks that throw unchecked exceptions.

A complete treatment of the Executor Framework is beyond the scope of this book, but the interested reader is directed to *Java Concurrency in Practice* [Goetz06].