

## CHAPTER 6

# Task Execution

Most concurrent applications are organized around the execution of *tasks*: abstract, discrete units of work. Dividing the work of an application into tasks simplifies program organization, facilitates error recovery by providing natural transaction boundaries, and promotes concurrency by providing a natural structure for parallelizing work.

### 6.1 Executing tasks in threads

The first step in organizing a program around task execution is identifying sensible *task boundaries*. Ideally, tasks are *independent* activities: work that doesn't depend on the state, result, or side effects of other tasks. Independence facilitates concurrency, as independent tasks can be executed in parallel if there are adequate processing resources. For greater flexibility in scheduling and load balancing tasks, each task should also represent a small fraction of your application's processing capacity.

Server applications should exhibit both *good throughput* and *good responsiveness* under normal load. Application providers want applications to support as many users as possible, so as to reduce provisioning costs per user; users want to get their response quickly. Further, applications should exhibit *graceful degradation* as they become overloaded, rather than simply falling over under heavy load. Choosing good task boundaries, coupled with a sensible *task execution policy* (see Section 6.2.2), can help achieve these goals.


Most server applications offer a natural choice of task boundary: individual client requests. Web servers, mail servers, file servers, EJB containers, and database servers all accept requests via network connections from remote clients. Using individual requests as task boundaries usually offers both independence and appropriate task sizing. For example, the result of submitting a message to a mail server is not affected by the other messages being processed at the same time, and handling a single message usually requires a very small percentage of the server's total capacity.

### 6.1.1 Executing tasks sequentially

There are a number of possible policies for scheduling tasks within an application, some of which exploit the potential for concurrency better than others. The simplest is to execute tasks sequentially in a single thread. `SingleThreadWebServer` in Listing 6.1 processes its tasks—HTTP requests arriving on port 80—sequentially. The details of the request processing aren't important; we're interested in characterizing the concurrency of various scheduling policies.

---

```
class SingleThreadWebServer {  
    public static void main(String[] args) throws IOException {  
        ServerSocket socket = new ServerSocket(80);  
        while (true) {  
            Socket connection = socket.accept();  
            handleRequest(connection);  
        }  
    }  
}
```



---

LISTING 6.1. Sequential web server.

`SingleThreadedWebServer` is simple and theoretically correct, but would perform poorly in production because it can handle only one request at a time. The main thread alternates between accepting connections and processing the associated request. While the server is handling a request, new connections must wait until it finishes the current request and calls `accept` again. This might work if request processing were so fast that `handleRequest` effectively returned immediately, but this doesn't describe any web server in the real world.

Processing a web request involves a mix of computation and I/O. The server must perform socket I/O to read the request and write the response, which can block due to network congestion or connectivity problems. It may also perform file I/O or make database requests, which can also block. In a single-threaded server, blocking not only delays completing the current request, but prevents pending requests from being processed at all. If one request blocks for an unusually long time, users might think the server is unavailable because it appears unresponsive. At the same time, resource utilization is poor, since the CPU sits idle while the single thread waits for its I/O to complete.

In server applications, sequential processing rarely provides either good throughput or good responsiveness. There are exceptions—such as when tasks are few and long-lived, or when the server serves a single client that makes only a single request at a time—but most server applications do not work this way.<sup>1</sup>

---

1. In some situations, sequential processing may offer a simplicity or safety advantage; most GUI frameworks process tasks sequentially using a single thread. We return to the sequential model in Chapter 9.

### 6.1.2 Explicitly creating threads for tasks

A more responsive approach is to create a new thread for servicing each request, as shown in `ThreadPerTaskWebServer` in Listing 6.2.

---

```
class ThreadPerTaskWebServer {
    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            new Thread(task).start();
        }
    }
}
```

---



LISTING 6.2. Web server that starts a new thread for each request.

`ThreadPerTaskWebServer` is similar in structure to the single-threaded version—the main thread still alternates between accepting an incoming connection and dispatching the request. The difference is that for each connection, the main loop creates a new thread to process the request instead of processing it within the main thread. This has three main consequences:

- Task processing is offloaded from the main thread, enabling the main loop to resume waiting for the next incoming connection more quickly. This enables new connections to be accepted before previous requests complete, improving responsiveness.
- Tasks can be processed in parallel, enabling multiple requests to be serviced simultaneously. This may improve throughput if there are multiple processors, or if tasks need to block for any reason such as I/O completion, lock acquisition, or resource availability.
- Task-handling code must be thread-safe, because it may be invoked concurrently for multiple tasks.

Under light to moderate load, the thread-per-task approach is an improvement over sequential execution. As long as the request arrival rate does not exceed the server's capacity to handle requests, this approach offers better responsiveness and throughput.

### 6.1.3 Disadvantages of unbounded thread creation

For production use, however, the thread-per-task approach has some practical drawbacks, especially when a large number of threads may be created:

**Thread lifecycle overhead.** Thread creation and teardown are not free. The actual overhead varies across platforms, but thread creation takes time, introducing latency into request processing, and requires some processing activity by the JVM and OS. If requests are frequent and lightweight, as in most server applications, creating a new thread for each request can consume significant computing resources.

**Resource consumption.** Active threads consume system resources, especially memory. When there are more runnable threads than available processors, threads sit idle. Having many idle threads can tie up a lot of memory, putting pressure on the garbage collector, and having many threads competing for the CPUs can impose other performance costs as well. If you have enough threads to keep all the CPUs busy, creating more threads won't help and may even hurt.

**Stability.** There is a limit on how many threads can be created. The limit varies by platform and is affected by factors including JVM invocation parameters, the requested stack size in the Thread constructor, and limits on threads placed by the underlying operating system.<sup>2</sup> When you hit this limit, the most likely result is an `OutOfMemoryError`. Trying to recover from such an error is very risky; it is far easier to structure your program to avoid hitting this limit.

Up to a certain point, more threads can improve throughput, but beyond that point creating more threads just slows down your application, and creating one thread too many can cause your entire application to crash horribly. The way to stay out of danger is to place some bound on how many threads your application creates, and to test your application thoroughly to ensure that, even when this bound is reached, it does not run out of resources.

The problem with the thread-per-task approach is that nothing places any limit on the number of threads created except the rate at which remote users can throw HTTP requests at it. Like other concurrency hazards, unbounded thread creation may *appear* to work just fine during prototyping and development, with problems surfacing only when the application is deployed and under heavy load. So a malicious user, or enough ordinary users, can make your web server crash if the traffic load ever reaches a certain threshold. For a server application that is supposed to provide high availability and graceful degradation under load, this is a serious failing.

---

2. On 32-bit machines, a major limiting factor is address space for thread stacks. Each thread maintains two execution stacks, one for Java code and one for native code. Typical JVM defaults yield a combined stack size of around half a megabyte. (You can change this with the `-Xss` JVM flag or through the Thread constructor.) If you divide the per-thread stack size into  $2^{32}$ , you get a limit of a few thousands or tens of thousands of threads. Other factors, such as OS limitations, may impose stricter limits.

## 6.2 The Executor framework

Tasks are logical units of work, and threads are a mechanism by which tasks can run asynchronously. We've examined two policies for executing tasks using threads—execute tasks sequentially in a single thread, and execute each task in its own thread. Both have serious limitations: the sequential approach suffers from poor responsiveness and throughput, and the thread-per-task approach suffers from poor resource management.

In Chapter 5, we saw how to use *bounded queues* to prevent an overloaded application from running out of memory. *Thread pools* offer the same benefit for thread management, and `java.util.concurrent` provides a flexible thread pool implementation as part of the Executor framework. The primary abstraction for task execution in the Java class libraries is *not* `Thread`, but `Executor`, shown in Listing 6.3.

---

```
public interface Executor {  
    void execute(Runnable command);  
}
```

---

LISTING 6.3. Executor interface.

`Executor` may be a simple interface, but it forms the basis for a flexible and powerful framework for asynchronous task execution that supports a wide variety of task execution policies. It provides a standard means of decoupling *task submission* from *task execution*, describing tasks with `Runnable`. The `Executor` implementations also provide lifecycle support and hooks for adding statistics gathering, application management, and monitoring.

`Executor` is based on the producer-consumer pattern, where activities that submit tasks are the producers (producing units of work to be done) and the threads that execute tasks are the consumers (consuming those units of work). *Using an Executor is usually the easiest path to implementing a producer-consumer design in your application.*

### 6.2.1 Example: web server using Executor

Building a web server with an `Executor` is easy. `TaskExecutionWebServer` in Listing 6.4 replaces the hard-coded thread creation with an `Executor`. In this case, we use one of the standard `Executor` implementations, a fixed-size thread pool with 100 threads.

In `TaskExecutionWebServer`, submission of the request-handling task is decoupled from its execution using an `Executor`, and its behavior can be changed merely by substituting a different `Executor` implementation. Changing `Executor` implementations or configuration is far less invasive than changing the way tasks are submitted; `Executor` configuration is generally a one-time event and can easily be exposed for deployment-time configuration, whereas task submission code tends to be strewn throughout the program and harder to expose.

---

```

class TaskExecutionWebServer {
    private static final int NTHREADS = 100;
    private static final Executor exec
        = Executors.newFixedThreadPool(NTHREADS);

    public static void main(String[] args) throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (true) {
            final Socket connection = socket.accept();
            Runnable task = new Runnable() {
                public void run() {
                    handleRequest(connection);
                }
            };
            exec.execute(task);
        }
    }
}

```

---

LISTING 6.4. Web server using a thread pool.

We can easily modify `TaskExecutionWebServer` to behave like `ThreadPerTaskWebServer` by substituting an `Executor` that creates a new thread for each request. Writing such an `Executor` is trivial, as shown in `ThreadPerTaskExecutor` or in Listing 6.5.

---

```

public class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    }
}

```

---

LISTING 6.5. Executor that starts a new thread for each task.

Similarly, it is also easy to write an `Executor` that would make `TaskExecutionWebServer` behave like the single-threaded version, executing each task synchronously before returning from `execute`, as shown in `WithinThreadExecutor` in Listing 6.6.

### 6.2.2 Execution policies

The value of decoupling submission from execution is that it lets you easily specify, and subsequently change without great difficulty, the *execution policy* for a given class of tasks. An execution policy specifies the “what, where, when, and how” of task execution, including:

---

```
public class WithinThreadExecutor implements Executor {  
    public void execute(Runnable r) {  
        r.run();  
    };  
}
```

---

LISTING 6.6. Executor that executes tasks synchronously in the calling thread.

- In what thread will tasks be executed?
- In what order should tasks be executed (FIFO, LIFO, priority order)?
- How many tasks may execute concurrently?
- How many tasks may be queued pending execution?
- If a task has to be rejected because the system is overloaded, which task should be selected as the victim, and how should the application be notified?
- What actions should be taken before or after executing a task?

Execution policies are a resource management tool, and the optimal policy depends on the available computing resources and your quality-of-service requirements. By limiting the number of concurrent tasks, you can ensure that the application does not fail due to resource exhaustion or suffer performance problems due to contention for scarce resources.<sup>3</sup> Separating the specification of execution policy from task submission makes it practical to select an execution policy at deployment time that is matched to the available hardware.

Whenever you see code of the form:

```
new Thread(runnable).start()
```

and you think you might at some point want a more flexible execution policy, seriously consider replacing it with the use of an Executor.

### 6.2.3 Thread pools

A thread pool, as its name suggests, manages a homogeneous pool of worker threads. A thread pool is tightly bound to a *work queue* holding tasks waiting to be executed. Worker threads have a simple life: request the next task from the work queue, execute it, and go back to waiting for another task.

---

3. This is analogous to one of the roles of a transaction monitor in an enterprise application: it can throttle the rate at which transactions are allowed to proceed so as not to exhaust or overstress limited resources.

Executing tasks in pool threads has a number of advantages over the thread-per-task approach. Reusing an existing thread instead of creating a new one amortizes thread creation and teardown costs over multiple requests. As an added bonus, since the worker thread often already exists at the time the request arrives, the latency associated with thread creation does not delay task execution, thus improving responsiveness. By properly tuning the size of the thread pool, you can have enough threads to keep the processors busy while not having so many that your application runs out of memory or thrashes due to competition among threads for resources.

The class library provides a flexible thread pool implementation along with some useful predefined configurations. You can create a thread pool by calling one of the static factory methods in `Executors`:

**`newFixedThreadPool`**. A fixed-size thread pool creates threads as tasks are submitted, up to the maximum pool size, and then attempts to keep the pool size constant (adding new threads if a thread dies due to an unexpected `Exception`).

**`newCachedThreadPool`**. A cached thread pool has more flexibility to reap idle threads when the current size of the pool exceeds the demand for processing, and to add new threads when demand increases, but places no bounds on the size of the pool.

**`newSingleThreadExecutor`**. A single-threaded executor creates a single worker thread to process tasks, replacing it if it dies unexpectedly. Tasks are guaranteed to be processed sequentially according to the order imposed by the task queue (FIFO, LIFO, priority order).<sup>4</sup>

**`newScheduledThreadPool`**. A fixed-size thread pool that supports delayed and periodic task execution, similar to `Timer`. (See Section 6.2.5.)

The `newFixedThreadPool` and `newCachedThreadPool` factories return instances of the general-purpose `ThreadPoolExecutor`, which can also be used directly to construct more specialized executors. We discuss thread pool configuration options in depth in Chapter 8.

The web server in `TaskExecutionWebServer` uses an `Executor` with a bounded pool of worker threads. Submitting a task with `execute` adds the task to the work queue, and the worker threads repeatedly dequeue tasks from the work queue and execute them.

Switching from a thread-per-task policy to a pool-based policy has a big effect on application stability: the web server will no longer fail under heavy load.<sup>5</sup>

---

4. Single-threaded executors also provide sufficient internal synchronization to guarantee that any memory writes made by tasks are visible to subsequent tasks; this means that objects can be safely confined to the “task thread” even though that thread may be replaced with another from time to time.

5. While the server may not fail due to the creation of too many threads, if the task arrival rate exceeds the task service rate for long enough it is still possible (just harder) to run out of memory because of the growing queue of `Runnable`s awaiting execution. This can be addressed within the `Executor` framework by using a bounded work queue—see Section 8.3.2.



It also degrades more gracefully, since it does not create thousands of threads that compete for limited CPU and memory resources. And using an Executor opens the door to all sorts of additional opportunities for tuning, management, monitoring, logging, error reporting, and other possibilities that would have been far more difficult to add without a task execution framework.

#### 6.2.4 Executor lifecycle

We've seen how to create an Executor but not how to shut one down. An Executor implementation is likely to create threads for processing tasks. But the JVM can't exit until all the (nondaemon) threads have terminated, so failing to shut down an Executor could prevent the JVM from exiting.

Because an Executor processes tasks asynchronously, at any given time the state of previously submitted tasks is not immediately obvious. Some may have completed, some may be currently running, and others may be queued awaiting execution. In shutting down an application, there is a spectrum from graceful shutdown (finish what you've started but don't accept any new work) to abrupt shutdown (turn off the power to the machine room), and various points in between. Since Executors provide a service to applications, they should be able to be shut down as well, both gracefully and abruptly, and feed back information to the application about the status of tasks that were affected by the shutdown.

To address the issue of execution service lifecycle, the `ExecutorService` interface extends `Executor`, adding a number of methods for lifecycle management (as well as some convenience methods for task submission). The lifecycle management methods of `ExecutorService` are shown in Listing 6.7.

---

```
public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException;
    // ... additional convenience methods for task submission
}
```

---

LISTING 6.7. Lifecycle methods in `ExecutorService`.

The lifecycle implied by `ExecutorService` has three states—*running*, *shutting down*, and *terminated*. `ExecutorServices` are initially created in the *running* state. The `shutdown` method initiates a graceful shutdown: no new tasks are accepted but previously submitted tasks are allowed to complete—including those that have not yet begun execution. The `shutdownNow` method initiates an abrupt shutdown: it attempts to cancel outstanding tasks and does not start any tasks that are queued but not begun.

Tasks submitted to an `ExecutorService` after it has been shut down are handled by the *rejected execution handler* (see Section 8.3.3), which might silently dis-

card the task or might cause `execute` to throw the unchecked `RejectedExecutionException`. Once all tasks have completed, the `ExecutorService` transitions to the *terminated* state. You can wait for an `ExecutorService` to reach the terminated state with `awaitTermination`, or poll for whether it has yet terminated with `isTerminated`. It is common to follow shutdown immediately by `awaitTermination`, creating the effect of synchronously shutting down the `ExecutorService`. (Executor shutdown and task cancellation are covered in more detail in Chapter 7.)

`LifecycleWebServer` in Listing 6.8 extends our web server with lifecycle support. It can be shut down in two ways: programmatically by calling `stop`, and through a client request by sending the web server a specially formatted HTTP request.

---

```
class LifecycleWebServer {
    private final ExecutorService exec = ...;

    public void start() throws IOException {
        ServerSocket socket = new ServerSocket(80);
        while (!exec.isShutdown()) {
            try {
                final Socket conn = socket.accept();
                exec.execute(new Runnable() {
                    public void run() { handleRequest(conn); }
                });
            } catch (RejectedExecutionException e) {
                if (!exec.isShutdown())
                    log("task submission rejected", e);
            }
        }
    }

    public void stop() { exec.shutdown(); }

    void handleRequest(Socket connection) {
        Request req = readRequest(connection);
        if (isShutdownRequest(req))
            stop();
        else
            dispatchRequest(req);
    }
}
```

---

LISTING 6.8. Web server with shutdown support.

### 6.2.5 Delayed and periodic tasks

The `Timer` facility manages the execution of deferred (“run this task in 100 ms”) and periodic (“run this task every 10 ms”) tasks. However, `Timer` has some drawbacks, and `ScheduledThreadPoolExecutor` should be thought of as its replacement.<sup>6</sup> You can construct a `ScheduledThreadPoolExecutor` through its constructor or through the `newScheduledThreadPool` factory.

A `Timer` creates only a single thread for executing timer tasks. If a timer task takes too long to run, the timing accuracy of other `TimerTasks` can suffer. If a recurring `TimerTask` is scheduled to run every 10 ms and another `TimerTask` takes 40 ms to run, the recurring task either (depending on whether it was scheduled at fixed rate or fixed delay) gets called four times in rapid succession after the long-running task completes, or “misses” four invocations completely. Scheduled thread pools address this limitation by letting you provide multiple threads for executing deferred and periodic tasks.

Another problem with `Timer` is that it behaves poorly if a `TimerTask` throws an unchecked exception. The `Timer` thread doesn’t catch the exception, so an unchecked exception thrown from a `TimerTask` terminates the timer thread. `Timer` also doesn’t resurrect the thread in this situation; instead, it erroneously assumes the entire `Timer` was cancelled. In this case, `TimerTasks` that are already scheduled but not yet executed are never run, and new tasks cannot be scheduled. (This problem, called “thread leakage” is described in Section 7.3, along with techniques for avoiding it.)

`OutOfTime` in Listing 6.9 illustrates how a `Timer` can become confused in this manner and, as confusion loves company, how the `Timer` shares its confusion with the next hapless caller that tries to submit a `TimerTask`. You might expect the program to run for six seconds and exit, but what actually happens is that it terminates after one second with an `IllegalStateException` whose message text is “Timer already cancelled”. `ScheduledThreadPoolExecutor` deals properly with ill-behaved tasks; there is little reason to use `Timer` in Java 5.0 or later.

If you need to build your own scheduling service, you may still be able to take advantage of the library by using a `DelayQueue`, a `BlockingQueue` implementation that provides the scheduling functionality of `ScheduledThreadPoolExecutor`. A `DelayQueue` manages a collection of `Delayed` objects. A `Delayed` has a delay time associated with it: `DelayQueue` lets you take an element only if its delay has expired. Objects are returned from a `DelayQueue` ordered by the time associated with their delay.

## 6.3 Finding exploitable parallelism

The `Executor` framework makes it easy to specify an execution policy, but in order to use an `Executor`, you have to be able to describe your task as a `Runnable`. In most server applications, there is an obvious task boundary: a single client request. But sometimes good task boundaries are not quite so obvious, as

---

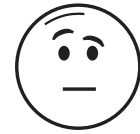
6. `Timer` does have support for scheduling based on absolute, not relative time, so that tasks can be sensitive to changes in the system clock; `ScheduledThreadPoolExecutor` supports only relative time.

---

```
public class OutOfTime {
    public static void main(String[] args) throws Exception {
        Timer timer = new Timer();
        timer.schedule(new ThrowTask(), 1);
        SECONDS.sleep(1);
        timer.schedule(new ThrowTask(), 1);
        SECONDS.sleep(5);
    }

    static class ThrowTask extends TimerTask {
        public void run() { throw new RuntimeException(); }
    }
}
```

---



LISTING 6.9. Class illustrating confusing Timer behavior.

in many desktop applications. There may also be exploitable parallelism within a single client request in server applications, as is sometimes the case in database servers. (For a further discussion of the competing design forces in choosing task boundaries, see [CPJ 4.4.1.1].)

In this section we develop several versions of a component that admit varying degrees of concurrency. Our sample component is the page-rendering portion of a browser application, which takes a page of HTML and renders it into an image buffer. To keep it simple, we assume that the HTML consists only of marked up text interspersed with image elements with pre-specified dimensions and URLs.

### 6.3.1 Example: sequential page renderer

The simplest approach is to process the HTML document sequentially. As text markup is encountered, render it into the image buffer; as image references are encountered, fetch the image over the network and draw it into the image buffer as well. This is easy to implement and requires touching each element of the input only once (it doesn't even require buffering the document), but is likely to annoy the user, who may have to wait a long time before all the text is rendered.

A less annoying but still sequential approach involves rendering the text elements first, leaving rectangular placeholders for the images, and after completing the initial pass on the document, going back and downloading the images and drawing them into the associated placeholder. This approach is shown in `SingleThreadRenderer` in Listing 6.10.

Downloading an image mostly involves waiting for I/O to complete, and during this time the CPU does little work. So the sequential approach may underutilize the CPU, and also makes the user wait longer than necessary to see the finished page. We can achieve better utilization and responsiveness by breaking the problem into independent tasks that can execute concurrently.

---

```
public class SingleThreadRenderer {  
    void renderPage(CharSequence source) {  
        renderText(source);  
        List<ImageData> imageData = new ArrayList<ImageData>();  
        for (ImageInfo imageInfo : scanForImageInfo(source))  
            imageData.add(imageInfo.downloadImage());  
        for (ImageData data : imageData)  
            renderImage(data);  
    }  
}
```

---



LISTING 6.10. Rendering page elements sequentially.

### 6.3.2 Result-bearing tasks: `Callable` and `Future`

The `Executor` framework uses `Runnable` as its basic task representation. `Runnable` is a fairly limiting abstraction; `run` cannot return a value or throw checked exceptions, although it can have side effects such as writing to a log file or placing a result in a shared data structure.

Many tasks are effectively deferred computations—executing a database query, fetching a resource over the network, or computing a complicated function. For these types of tasks, `Callable` is a better abstraction: it expects that the main entry point, `call`, will return a value and anticipates that it might throw an exception.<sup>7</sup> `Executors` includes several utility methods for wrapping other types of tasks, including `Runnable` and `java.security.PrivilegedAction`, with a `Callable`.

`Runnable` and `Callable` describe abstract computational tasks. Tasks are usually finite: they have a clear starting point and they eventually terminate. The lifecycle of a task executed by an `Executor` has four phases: *created*, *submitted*, *started*, and *completed*. Since tasks can take a long time to run, we also want to be able to cancel a task. In the `Executor` framework, tasks that have been submitted but not yet started can always be cancelled, and tasks that have started can sometimes be cancelled if they are responsive to interruption. Cancelling a task that has already completed has no effect. (Cancellation is covered in greater detail in Chapter 7.)

`Future` represents the lifecycle of a task and provides methods to test whether the task has completed or been cancelled, retrieve its result, and cancel the task. `Callable` and `Future` are shown in Listing 6.11. Implicit in the specification of `Future` is that task lifecycle can only move forwards, not backwards—just like the `ExecutorService` lifecycle. Once a task is completed, it stays in that state forever.

The behavior of `get` varies depending on the task state (not yet started, running, completed). It returns immediately or throws an `Exception` if the task has already completed, but if not it blocks until the task completes. If the task completes by throwing an exception, `get` rethrows it wrapped in an `Execution-`

---

7. To express a non-value-returning task with `Callable`, use `Callable<Void>`.

---

```
public interface Callable<V> {
    V call() throws Exception;
}

public interface Future<V> {
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
    V get() throws InterruptedException, ExecutionException,
        CancellationException;
    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException,
        CancellationException, TimeoutException;
}
```

---

LISTING 6.11. Callable and Future interfaces.

Exception; if it was cancelled, `get` throws `CancellationException`. If `get` throws `ExecutionException`, the underlying exception can be retrieved with `getCause`.

There are several ways to create a `Future` to describe a task. The `submit` methods in `ExecutorService` all return a `Future`, so that you can submit a `Runnable` or a `Callable` to an executor and get back a `Future` that can be used to retrieve the result or cancel the task. You can also explicitly instantiate a `FutureTask` for a given `Runnable` or `Callable`. (Because `FutureTask` implements `Runnable`, it can be submitted to an `Executor` for execution or executed directly by calling its `run` method.)

As of Java 6, `ExecutorService` implementations can override `newTaskFor` in `AbstractExecutorService` to control instantiation of the `Future` corresponding to a submitted `Callable` or `Runnable`. The default implementation just creates a new `FutureTask`, as shown in Listing 6.12.

---

```
protected <T> RunnableFuture<T> newTaskFor(Callable<T> task) {
    return new FutureTask<T>(task);
}
```

---

LISTING 6.12. Default implementation of `newTaskFor` in `ThreadPoolExecutor`.

Submitting a `Runnable` or `Callable` to an `Executor` constitutes a safe publication (see Section 3.5) of the `Runnable` or `Callable` from the submitting thread to the thread that will eventually execute the task. Similarly, setting the result value for a `Future` constitutes a safe publication of the result from the thread in which it was computed to any thread that retrieves it via `get`.

### 6.3.3 Example: page renderer with Future

As a first step towards making the page renderer more concurrent, let's divide it into two tasks, one that renders the text and one that downloads all the images. (Because one task is largely CPU-bound and the other is largely I/O-bound, this approach may yield improvements even on single-CPU systems.)

`Callable` and `Future` can help us express the interaction between these cooperating tasks. In `FutureRenderer` in Listing 6.13, we create a `Callable` to download all the images, and submit it to an `ExecutorService`. This returns a `Future` describing the task's execution; when the main task gets to the point where it needs the images, it waits for the result by calling `Future.get`. If we're lucky, the results will already be ready by the time we ask; otherwise, at least we got a head start on downloading the images.

The state-dependent nature of `get` means that the caller need not be aware of the state of the task, and the safe publication properties of task submission and result retrieval make this approach thread-safe. The exception handling code surrounding `Future.get` deals with two possible problems: that the task encountered an `Exception`, or the thread calling `get` was interrupted before the results were available. (See Sections 5.5.2 and 5.4.)

`FutureRenderer` allows the text to be rendered concurrently with downloading the image data. When all the images are downloaded, they are rendered onto the page. This is an improvement in that the user sees a result quickly and it exploits some parallelism, but we can do considerably better. There is no need for users to wait for *all* the images to be downloaded; they would probably prefer to see individual images drawn as they become available.

### 6.3.4 Limitations of parallelizing heterogeneous tasks

In the last example, we tried to execute two different types of tasks in parallel—downloading the images and rendering the page. But obtaining significant performance improvements by trying to parallelize sequential heterogeneous tasks can be tricky.

Two people can divide the work of cleaning the dinner dishes fairly effectively: one person washes while the other dries. However, assigning a different type of task to each worker does not scale well; if several more people show up, it is not obvious how they can help without getting in the way or significantly restructuring the division of labor. Without finding finer-grained parallelism among similar tasks, this approach will yield diminishing returns.

A further problem with dividing heterogeneous tasks among multiple workers is that the tasks may have disparate sizes. If you divide tasks *A* and *B* between two workers but *A* takes ten times as long as *B*, you've only speeded up the total process by 9%. Finally, dividing a task among multiple workers always involves some amount of coordination overhead; for the division to be worthwhile, this overhead must be more than compensated by productivity improvements due to parallelism.

`FutureRenderer` uses two tasks: one for rendering text and one for downloading the images. If rendering the text is much faster than downloading the images,


---

```
public class FutureRenderer {
    private final ExecutorService executor = ...;

    void renderPage(CharSequence source) {
        final List<ImageInfo> imageInfos = scanForImageInfo(source);
        Callable<List<ImageData>> task =
            new Callable<List<ImageData>>() {
                public List<ImageData> call() {
                    List<ImageData> result
                        = new ArrayList<ImageData>();
                    for (ImageInfo imageInfo : imageInfos)
                        result.add(imageInfo.downloadImage());
                    return result;
                }
            };

        Future<List<ImageData>> future = executor.submit(task);
        renderText(source);

        try {
            List<ImageData> imageData = future.get();
            for (ImageData data : imageData)
                renderImage(data);
        } catch (InterruptedException e) {
            // Re-assert the thread's interrupted status
            Thread.currentThread().interrupt();
            // We don't need the result, so cancel the task too
            future.cancel(true);
        } catch (ExecutionException e) {
            throw launderThrowable(e.getCause());
        }
    }
}
```



---

LISTING 6.13. Waiting for image download with Future.



as is entirely possible, the resulting performance is not much different from the sequential version, but the code is a lot more complicated. And the best we can do with two threads is speed things up by a factor of two. Thus, trying to increase concurrency by parallelizing heterogeneous activities can be a lot of work, and there is a limit to how much additional concurrency you can get out of it. (See Sections 11.4.2 and 11.4.3 for another example of the same phenomenon.)

The real performance payoff of dividing a program's workload into tasks comes when there are a large number of independent, *homogeneous* tasks that can be processed concurrently.

### 6.3.5 CompletionService: Executor meets BlockingQueue

If you have a batch of computations to submit to an Executor and you want to retrieve their results as they become available, you could retain the Future associated with each task and repeatedly poll for completion by calling `get` with a timeout of zero. This is possible, but tedious. Fortunately there is a better way: a *completion service*.

`CompletionService` combines the functionality of an Executor and a `BlockingQueue`. You can submit `Callable` tasks to it for execution and use the queue-like methods `take` and `poll` to retrieve completed results, packaged as `Futures`, as they become available. `ExecutorCompletionService` implements `CompletionService`, delegating the computation to an Executor.

The implementation of `ExecutorCompletionService` is quite straightforward. The constructor creates a `BlockingQueue` to hold the completed results. `FutureTask` has a `done` method that is called when the computation completes. When a task is submitted, it is wrapped with a `QueueingFuture`, a subclass of `FutureTask` that overrides `done` to place the result on the `BlockingQueue`, as shown in Listing 6.14. The `take` and `poll` methods delegate to the `BlockingQueue`, blocking if results are not yet available.

---

```
private class QueueingFuture<V> extends FutureTask<V> {
    QueueingFuture(Callable<V> c) { super(c); }
    QueueingFuture(Runnable t, V r) { super(t, r); }

    protected void done() {
        completionQueue.add(this);
    }
}
```

---

LISTING 6.14. `QueueingFuture` class used by `ExecutorCompletionService`.

### 6.3.6 Example: page renderer with CompletionService

We can use a `CompletionService` to improve the performance of the page renderer in two ways: shorter total runtime and improved responsiveness. We can create a separate task for downloading *each* image and execute them in a thread pool, turning the sequential download into a parallel one: this reduces the amount of time to download all the images. And by fetching results from the `CompletionService` and rendering each image as soon as it is available, we can give the user a more dynamic and responsive user interface. This implementation is shown in `Renderer` in Listing 6.15.

---

```
public class Renderer {
    private final ExecutorService executor;

    Renderer(ExecutorService executor) { this.executor = executor; }

    void renderPage(CharSequence source) {
        List<ImageInfo> info = scanForImageInfo(source);
        CompletionService<ImageData> completionService =
            new ExecutorCompletionService<ImageData>(executor);
        for (final ImageInfo imageInfo : info)
            completionService.submit(new Callable<ImageData>() {
                public ImageData call() {
                    return imageInfo.downloadImage();
                }
            });

        renderText(source);

        try {
            for (int t = 0, n = info.size(); t < n; t++) {
                Future<ImageData> f = completionService.take();
                ImageData imageData = f.get();
                renderImage(imageData);
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } catch (ExecutionException e) {
            throw launderThrowable(e.getCause());
        }
    }
}
```

---

LISTING 6.15. Using `CompletionService` to render page elements as they become available.

Multiple `ExecutorCompletionServices` can share a single `Executor`, so it is

perfectly sensible to create an `ExecutorCompletionService` that is private to a particular computation while sharing a common `Executor`. When used in this way, a `CompletionService` acts as a handle for a batch of computations in much the same way that a `Future` acts as a handle for a single computation. By remembering how many tasks were submitted to the `CompletionService` and counting how many completed results are retrieved, you can know when all the results for a given batch have been retrieved, even if you use a shared `Executor`.

### 6.3.7 Placing time limits on tasks

Sometimes, if an activity does not complete within a certain amount of time, the result is no longer needed and the activity can be abandoned. For example, a web application may fetch its advertisements from an external ad server, but if the ad is not available within two seconds, it instead displays a default advertisement so that ad unavailability does not undermine the site's responsiveness requirements. Similarly, a portal site may fetch data in parallel from multiple data sources, but may be willing to wait only a certain amount of time for data to be available before rendering the page without it.

The primary challenge in executing tasks within a time budget is making sure that you don't wait longer than the time budget to get an answer or find out that one is not forthcoming. The timed version of `Future.get` supports this requirement: it returns as soon as the result is ready, but throws `TimeoutException` if the result is not ready within the timeout period.

A secondary problem when using timed tasks is to stop them when they run out of time, so they do not waste computing resources by continuing to compute a result that will not be used. This can be accomplished by having the task strictly manage its own time budget and abort if it runs out of time, or by cancelling the task if the timeout expires. Again, `Future` can help; if a timed `get` completes with a `TimeoutException`, you can cancel the task through the `Future`. If the task is written to be cancellable (see Chapter 7), it can be terminated early so as not to consume excessive resources. This technique is used in Listings 6.13 and 6.16.

Listing 6.16 shows a typical application of a timed `Future.get`. It generates a composite web page that contains the requested content plus an advertisement fetched from an ad server. It submits the ad-fetching task to an executor, computes the rest of the page content, and then waits for the ad until its time budget runs out.<sup>8</sup> If the `get` times out, it cancels<sup>9</sup> the ad-fetching task and uses a default advertisement instead.

### 6.3.8 Example: a travel reservations portal

The time-budgeting approach in the previous section can be easily generalized to an arbitrary number of tasks. Consider a travel reservation portal: the user en-

---

8. The timeout passed to `get` is computed by subtracting the current time from the deadline; this may in fact yield a negative number, but all the timed methods in `java.util.concurrent` treat negative timeouts as zero, so no extra code is needed to deal with this case.

9. The `true` parameter to `Future.cancel` means that the task thread can be interrupted if the task is currently running; see Chapter 7.

---

```

Page renderPageWithAd() throws InterruptedException {
    long endNanos = System.nanoTime() + TIME_BUDGET;
    Future<Ad> f = exec.submit(new FetchAdTask());
    // Render the page while waiting for the ad
    Page page = renderPageBody();
    Ad ad;
    try {
        // Only wait for the remaining time budget
        long timeLeft = endNanos - System.nanoTime();
        ad = f.get(timeLeft, NANOSECONDS);
    } catch (ExecutionException e) {
        ad = DEFAULT_AD;
    } catch (TimeoutException e) {
        ad = DEFAULT_AD;
        f.cancel(true);
    }
    page.setAd(ad);
    return page;
}

```

---

LISTING 6.16. Fetching an advertisement with a time budget.

ters travel dates and requirements and the portal fetches and displays bids from a number of airlines, hotels or car rental companies. Depending on the company, fetching a bid might involve invoking a web service, consulting a database, performing an EDI transaction, or some other mechanism. Rather than have the response time for the page be driven by the slowest response, it may be preferable to present only the information available within a given time budget. For providers that do not respond in time, the page could either omit them completely or display a placeholder such as “Did not hear from Air Java in time.”

Fetching a bid from one company is independent of fetching bids from another, so fetching a single bid is a sensible task boundary that allows bid retrieval to proceed concurrently. It would be easy enough to create  $n$  tasks, submit them to a thread pool, retain the Futures, and use a timed get to fetch each result sequentially via its Future, but there is an even easier way—`invokeAll`.

Listing 6.17 uses the timed version of `invokeAll` to submit multiple tasks to an `ExecutorService` and retrieve the results. The `invokeAll` method takes a collection of tasks and returns a collection of Futures. The two collections have identical structures; `invokeAll` adds the Futures to the returned collection in the order imposed by the task collection’s iterator, thus allowing the caller to associate a Future with the `Callable` it represents. The timed version of `invokeAll` will return when all the tasks have completed, the calling thread is interrupted, or the timeout expires. Any tasks that are not complete when the timeout expires are cancelled. On return from `invokeAll`, each task will have either completed normally or been cancelled; the client code can call `get` or `isCancelled` to find

out which.

## Summary

Structuring applications around the execution of *tasks* can simplify development and facilitate concurrency. The Executor framework permits you to decouple task submission from execution policy and supports a rich variety of execution policies; whenever you find yourself creating threads to perform tasks, consider using an Executor instead. To maximize the benefit of decomposing an application into tasks, you must identify sensible task boundaries. In some applications, the obvious task boundaries work well, whereas in others some analysis may be required to uncover finer-grained exploitable parallelism.

---

```
private class QuoteTask implements Callable<TravelQuote> {
    private final TravelCompany company;
    private final TravelInfo travelInfo;
    ...
    public TravelQuote call() throws Exception {
        return company.solicitQuote(travelInfo);
    }
}

public List<TravelQuote> getRankedTravelQuotes(
    TravelInfo travelInfo, Set<TravelCompany> companies,
    Comparator<TravelQuote> ranking, long time, TimeUnit unit)
    throws InterruptedException {
    List<QuoteTask> tasks = new ArrayList<QuoteTask>();
    for (TravelCompany company : companies)
        tasks.add(new QuoteTask(company, travelInfo));

    List<Future<TravelQuote>> futures =
        exec.invokeAll(tasks, time, unit);

    List<TravelQuote> quotes =
        new ArrayList<TravelQuote>(tasks.size());
    Iterator<QuoteTask> taskIter = tasks.iterator();
    for (Future<TravelQuote> f : futures) {
        QuoteTask task = taskIter.next();
        try {
            quotes.add(f.get());
        } catch (ExecutionException e) {
            quotes.add(task.getFailureQuote(e.getCause()));
        } catch (CancellationException e) {
            quotes.add(task.getTimeoutQuote(e));
        }
    }

    Collections.sort(quotes, ranking);
    return quotes;
}
```

---

LISTING 6.17. Requesting travel quotes under a time budget.