SAMPLE CHAPTER

Lambdas, streams, functional and reactive programming

Modern Java IN ACTION

Raoul-Gabriel Urma
Mario Fusco
Alan Mycroft

www.itbook.store/books/9781617293566

Modern Java in Action

by Raoul-Gabriel Urma, Mario Fusco,
and Alan Mycroft

Chapter 15

Copyright 2018 Manning Publications

www.itbook.store/books/9781617293566

brief contents

PART 1 FUNDAMENTALS 1

1 ■ Java 8, 9, 10, and 11: what's happening? 3
2 ■ Passing code with behavior parameterization 26
3 ■ Lambda expressions 42

PART 2 FUNCTIONAL-STYLE DATA PROCESSING WITH STREAMS ... 79

4 ■ Introducing streams 81
5 ■ Working with streams 98
6 ■ Collecting data with streams 134
7 ■ Parallel data processing and performance 172

PART 3 EFFECTIVE PROGRAMMING WITH STREAMS AND LAMBDAS...199

8 ■ Collection API enhancements 201
9 ■ Refactoring, testing, and debugging 216
10 ■ Domain-specific languages using lambdas 239

www.itbook.store/books/9781617293566

PART 4 EVERYDAY JAVA 273

11 ■ Using Optional as a better alternative to null 275
12 ■ New Date and Time API 297
13 ■ Default methods 314
14 ■ The Java Module System 333

PART 5 ENHANCED JAVA CONCURRENCY 355

15 ■ Concepts behind CompletableFuture and reactive programming 357
16 ■ CompletableFuture: composable asynchronous programming 387
17 ■ Reactive programming 416

PART 6 FUNCTIONAL PROGRAMMING AND FUTURE JAVA EVOLUTION 443

18 ■ Thinking functionally 445
19 ■ Functional programming techniques 460
20 ■ Blending OOP and FP: Comparing Java and Scala 485
21 ■ Conclusions and where next for Java 500

www.itbook.store/books/9781617293566

15 Concepts behind CompletableFuture and reactive programming

This chapter covers

- Threads, Futures, and the evolutionary forces causing Java to support richer concurrency APIs
- Asynchronous APIs
- The boxes-and-channels view of concurrent computing
- CompletableFuture combinators to connect boxes dynamically
- The publish-subscribe protocol that forms the basis of the Java 9 Flow API for reactive programming
- Reactive programming and reactive systems

In recent years, two trends are obliging developers to rethink the way software is written. The first trend is related to the hardware on which applications run, and the second trend concerns how applications are structured (particularly how they interact). We discussed the effect of the hardware trend in chapter 7. We noted that since the advent of multicore processors, the most effective way to speed your applications is to write software that can fully exploit multicore processors. You saw that

www.itbook.store/books/9781617293566

you can split large tasks and make each subtask run in parallel with the others. You also learned how the fork/join framework (available since Java 7) and parallel streams (new in Java 8) allow you to accomplish this task in a simpler, more effective way than working directly with threads.

The second trend reflects the increasing availability and use by applications of Internet services. The adoption of microservices architecture, for example, has grown over the past few years. Instead of being one monolithic application, your application is subdivided into smaller services. The coordination of these smaller services requires increased network communication. Similarly, many more internet services are accessible through public APIs, made available by known providers such as Google (localization information), Facebook (social information), and Twitter (news). Nowadays, it's relatively rare to develop a website or a network application that works in total isolation. It's far more likely that your next web application will be a mashup, using content from multiple sources and aggregating it to ease your users' lives.

You may want to build a website that collects and summarizes social-media sentiment on a given topic to your French users. To do so, you could use the Facebook or Twitter API to find trending comments about that topic in many languages and rank the most relevant ones with your internal algorithms. Then you might use Google Translate to translate the comments into French or use Google Maps to geolocate their authors, aggregate all this information, and display it on your website.

If any of these external network services are slow to respond, of course, you'll want to provide partial results to your users, perhaps showing your text results alongside a generic map with a question mark in it instead of showing a blank screen until the map server responds or times out. Figure 15.1 illustrates how this style of *mashup* application interacts with remote services.

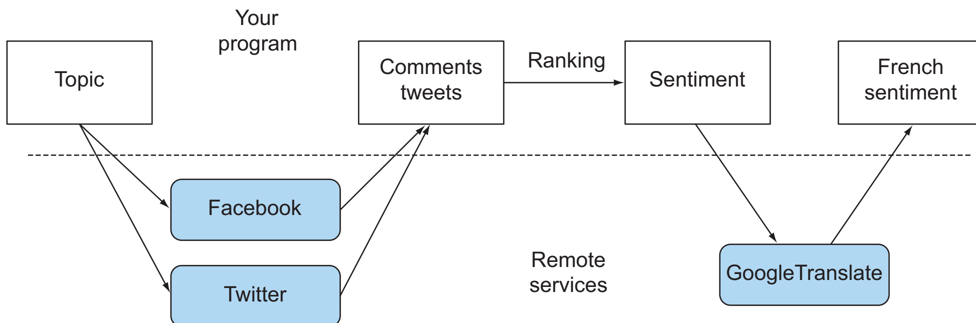

Figure 15.1 A typical mashup application

To implement applications like this, you have to contact multiple web services across the Internet. But you don't want to block your computations and waste billions of precious clock cycles of your CPU waiting for an answer from these services. You

www.itbook.store/books/9781617293566

shouldn't have to wait for data from Facebook before processing the data coming from Twitter, for example.

This situation represents the other side of the multitask-programming coin. The fork/join framework and parallel streams, discussed in chapter 7, are valuable tools for parallelism; they divide a task into multiple subtasks and perform those subtasks in parallel on different cores, CPUs, or even machines.

Conversely, when you're dealing with concurrency instead of parallelism, or when your main goal is to perform several loosely related tasks on the same CPUs, keeping their cores as busy as possible to maximize the throughput of your application, you want to avoid blocking a thread and wasting its computational resources while waiting (potentially for quite a while) for a result from a remote service or from interrogating a database.

Java offers two main tool sets for such circumstances. First, as you'll see in chapters 16 and 17, the `Future` interface, and particularly its Java 8 `CompletableFuture` implementation, often provide simple and effective solutions (chapter 16). More recently, Java 9 added the idea of reactive programming, built around the idea of the so-called publish-subscribe protocol via the `Flow` API, which offers more sophisticated programming approaches (chapter 17).

Figure 15.2 illustrates the difference between concurrency and parallelism. Concurrency is a programming property (overlapped execution) that can occur even for a single-core machine, whereas parallelism is a property of execution hardware (simultaneous execution).

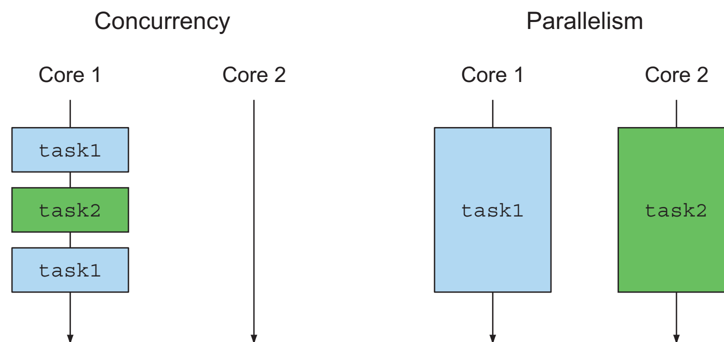

Figure 15.2 Concurrency versus parallelism

The rest of this chapter explains the fundamental ideas underpinning Java's new CompletableFuture and Flow APIs.

We start by explaining the Java evolution of concurrency, including Threads, and higher-level abstractions, including Thread Pools and Futures (section 15.1). We note that chapter 7 dealt with mainly using parallelism in looplike programs. Section 15.2 explores how you can better exploit concurrency for method calls. Section 15.3 gives you a diagrammatic way to see parts of programs as boxes that communicate over

www.itbook.store/books/9781617293566

> **Guidance for the reader**
>
> This chapter contains little real-life Java code. We suggest that readers who want to see only code skip to chapters 16 and 17. On the other hand, as we've all discovered, code that implements unfamiliar ideas can be hard to understand. Therefore, we use simple functions and include diagrams to explain the big-picture ideas, such as the publish-subscribe protocol behind the Flow API capturing reactive programming.

channels. Section 15.4 and section 15.5 look at the CompletableFuture and reactive-programming principles in Java 8 and 9. Finally, section 15.6 explains the difference between a reactive system and reactive programming.

We exemplify most of the concepts with a running example showing how to calculate expressions such as `f(x)+g(x)` and then return, or print, the result by using various Java concurrency features—assuming that `f(x)` and `g(x)` are long-running computations.

15.1 Evolving Java support for expressing concurrency

Java has evolved considerably in its support for concurrent programming, largely reflecting the changes in hardware, software systems, and programming concepts over the past 20 years. Summarizing this evolution can help you understand the reason for the new additions and their roles in programming and system design.

Initially, Java had locks (via `synchronized` classes and methods), `Runnables` and `Threads`. In 2004, Java 5 introduced the `java.util.concurrent` package, which supported more expressive concurrency, particularly the `ExecutorService`[1] interface (which decoupled task submission from thread execution), as well as `Callable<T>` and `Future<T>`, which produced higher-level and result-returning variants of `Runnable` and `Thread` and used generics (also introduced in Java 5). `ExecutorServices` can execute both `Runnables` and `Callables`. These features facilitated parallel programming on the multicore CPUs that started to appear the following year. To be honest, nobody enjoyed working with threads directly!

Later versions of Java continued to enhance concurrency support, as it became increasingly demanded by programmers who needed to program multicore CPUs effectively. As you saw in chapter 7, Java 7 added `java.util.concurrent.RecursiveTask` to support fork/join implementation of divide-and-conquer algorithms, and Java 8 added support for `Streams` and their parallel processing (building on the newly added support for lambdas).

Java further enriched its concurrency features by providing support for *composing* Futures (via the Java 8 `CompletableFuture` implementation of `Future`, section 15.4 and chapter 16), and Java 9, provided explicit support for distributed asynchronous programming. These APIs give you a mental model and toolkit for building the sort of mashup application mentioned in the introduction to this chapter. There the

[1] The `ExecutorService` interface extends the `Executor` interface with the `submit` method to run a `Callable`; the `Executor` interface merely has an `execute` method for `Runnables`.

www.itbook.store/books/9781617293566

application worked by contacting various web services and combining their information in real time for a user or to expose it as a further web service. This process is called *reactive programming*, and Java 9 provides support for it via the *publish-subscribe protocol* (specified by the `java.util.concurrent.Flow` interface; see section 15.5 and chapter 17). A key concept of `CompletableFuture` and `java.util.concurrent.Flow` is to provide programming structures that enable independent tasks to execute concurrently wherever possible and in a way that easily exploits as much as possible of the parallelism provided by multicore or multiple machines.

15.1.1 Threads and higher-level abstractions

Many of us learned about threads and processes from a course on operating systems. A single-CPU computer can support multiple users because its operating system allocates a process to each user. The operating system gives these processes separate virtual address spaces so that two users feel like they're the only users of the computer. The operating system furthers this illusion by waking periodically to share the CPU among the processes. A process can request that the operating system allocate it one or more *threads*—processes that share an address space as their owning process and therefore can run tasks concurrently and cooperatively.

In a multicore setting, perhaps a single-user laptop running only one user process, a program can never fully exploit the computing power of the laptop unless it uses threads. Each core can be used for one or more processes or threads, but if your program doesn't use threads, it's effectively using only one of the processor cores.

Indeed, if you have a four-core CPU and can arrange for each core to continually do useful work, your program theoretically runs up to four times faster. (Overheads reduce this result somewhere, of course.) Given an array of numbers of size 1,000,000 storing the number of correct questions answered by students in an example, compare the program

```
long sum = 0;
for (int i = 0; i < 1_000_000; i++) {
    sum += stats[i];
}
```

running on a single thread, which worked fine in single-core days, with a version that creates four threads, with the first thread executing

```
long sum0 = 0;
for (int i = 0; i < 250_000; i++) {
    sum0 += stats[i];
}
```

and to the fourth thread executing

```
long sum3 = 0;
for (int i = 750_000; i < 1_000_000; i++) {
    sum3 += stats[i];
}
```

www.itbook.store/books/9781617293566

These four threads are complemented by the main program starting them in turn (`.start()` in Java), waiting for them to complete (`.join()`), and then computing

```
sum = sum0 + … + sum3;
```

The trouble is that doing this for each loop is tedious and error-prone. Also, what can you do for code that isn't a loop?

Chapter 7 showed how Java Streams can achieve this parallelism with little programmer effort by using internal iteration instead of external iteration (explicit loops):

```
sum = Arrays.stream(stats).parallel().sum();
```

The takeaway idea is that parallel Stream iteration is a higher-level concept than explicit use of threads. In other words, this use of Streams *abstracts* a given use pattern of threads. This abstraction into Streams is analogous to a design pattern, but with the benefit that much of the complexity is implemented inside the library rather than being boilerplate code. Chapter 7 also explained how to use `java.util.concurrent.RecursiveTask` support in Java 7 for the fork/join abstraction of threads to parallelize divide-and-conquer algorithms, providing a higher-level way to sum the array efficiently on a multicore machine.

Before looking at additional abstractions for threads, we visit the (Java 5) idea of `ExecutorServices` and the thread pools on which these further abstractions are built.

15.1.2 Executors and thread pools

Java 5 provided the Executor framework and the idea of thread pools as a higher-level idea capturing the power of threads, which allow Java programmers to decouple task submission from task execution.

PROBLEMS WITH THREADS

Java threads access operating-system threads directly. The problem is that operating-system threads are expensive to create and to destroy (involving interaction with page tables), and moreover, only a limited number exist. Exceeding the number of operating-system threads is likely to cause a Java application to crash mysteriously, so be careful not to leave threads running while continuing to create new ones.

The number of operating system (and Java) threads will significantly exceed the number of hardware threads[2], so all the hardware threads can be usefully occupied executing code even when some operating-system threads are blocked or sleeping. As an example, the 2016 Intel Core i7-6900K server processor has eight cores, each with two symmetric multiprocessing (SMP) hardware threads, leading to 16 hardware threads, and a server may contain several of these processors, consisting of perhaps 64 hardware threads. By contrast, a laptop may have only one or two hardware threads, so portable programs must avoid making assumptions about how many hardware threads

[2] We'd use the word *core* here, but CPUs like the Intel i7-6900K have multiple hardware threads per core, so the CPU can execute useful instructions even for short delays such as a cache miss.

www.itbook.store/books/9781617293566

are available. Contrarily, the optimum number of Java threads for a given program depends on the number of hardware cores available!

Thread pools and why they're better

The Java `ExecutorService` provides an interface where you can submit tasks and obtain their results later. The expected implementation uses a pool of threads, which can be created by one of the factory methods, such as the `newFixedThreadPool` method:

```
ExecutorService newFixedThreadPool(int nThreads)
```

This method creates an `ExecutorService` containing `nThreads` (often called *worker threads*) and stores them in a thread pool, from which unused threads are taken to run submitted tasks on a first-come, first-served basis. These threads are returned to the pool when their tasks terminate. One great outcome is that it's cheap to submit thousands of tasks to a thread pool while keeping the number of tasks to a hardware-appropriate number. Several configurations are possible, including the size of the queue, rejection policy, and priority for different tasks.

Note the wording: The programmer provides a *task* (a `Runnable` or a `Callable`), which is executed by a *thread*.

Thread pools and why they're worse

Thread pools are better than explicit thread manipulation in almost all ways, but you need to be aware of two "gotchas:"

- A thread pool with *k* threads can execute only *k* tasks concurrently. Any further task submissions are held in a queue and not allocated a thread until one of the existing tasks completes. This situation is generally good, in that it allows you to submit many tasks without accidentally creating an excessive number of threads, but you have to be wary of tasks that sleep or wait for I/O or network connections. In the context of blocking I/O, these tasks occupy worker threads but do no useful work while they're waiting. Try taking four hardware threads and a thread pool of size 5 and submitting 20 tasks to it (figure 15.3). You might expect that the tasks would run in parallel until all 20 have completed. But suppose that three of the first-submitted tasks sleep or wait for I/O. Then only two threads are available for the remaining 15 tasks, so you're getting only half the throughput you expected (and would have if you created the thread pool with eight threads instead). It's even possible to cause deadlock in a thread pool if earlier task submissions or already running tasks, need to wait for later task submissions, which is a typical use-pattern for Futures.

  The takeaway is to try to avoid submitting tasks that can block (sleep or wait for events) to thread pools, but you can't always do so in existing systems.
- Java typically waits for all threads to complete before allowing a return from `main` to avoid killing a thread executing vital code. Therefore, it's important in practice and as part of good hygiene to shut down every thread pool before exiting the program (because worker threads for this pool will have been created but

www.itbook.store/books/9781617293566

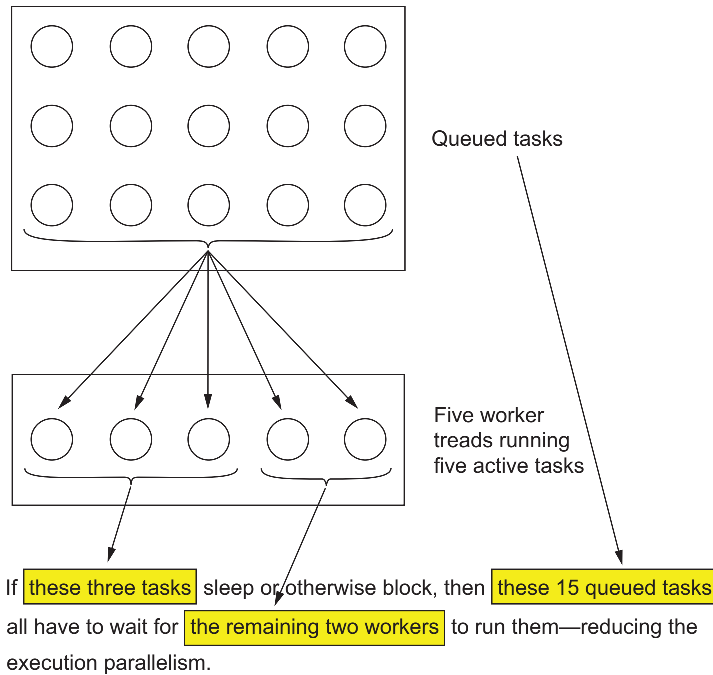

Figure 15.3 Sleeping tasks reduce the throughput of thread pools.

not terminated, as they're waiting for another task submission). In practice, it's common to have a long-running `ExecutorService` that manages an always-running Internet service.

Java does provide the `Thread.setDaemon` method to control this behavior, which we discuss in the next section.

15.1.3 Other abstractions of threads: non-nested with method calls

To explain why the forms of concurrency used in this chapter differ from those used in chapter 7 (parallel Stream processing and the fork/join framework), we'll note that the forms used in chapter 7 have one special property: whenever any task (or thread) is started within a method call, the same method call waits for it to complete before returning. In other words, thread creation and the matching `join()` happen in a way that nests properly within the call-return nesting of method calls. This idea, called *strict fork/join*, is depicted in figure 15.4.

It's relatively innocuous to have a more relaxed form of fork/join in which a spawned task escapes from an internal method call but is joined in an outer call, so that the interface provided to users still appears to be a normal call,[3] as shown in figure 15.5.

[3] Compare "Thinking Functionally" (chapter 18) in which we discuss having a side-effect-free interface to a method that internally uses side-effects!

www.itbook.store/books/9781617293566

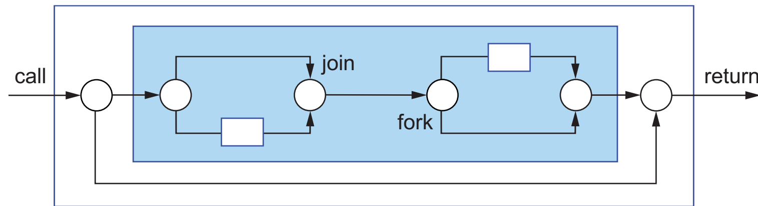

Figure 15.4 Strict fork/join. Arrows denote threads, circles represent forks and joins, and rectangles represent method calls and returns.

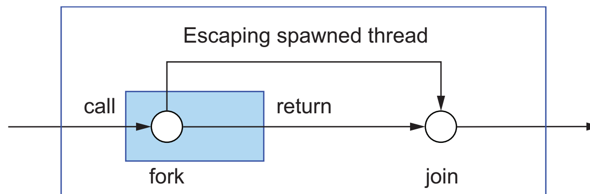

Figure 15.5 Relaxed fork/join

In this chapter, we focus on richer forms of concurrency in which threads created (or tasks spawned) by a user's method call may outlive the call, as shown in figure 15.6.

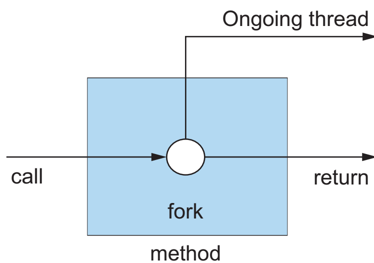

Figure 15.6 An asynchronous method

This type of method is often called an asynchronous method, particularly when the ongoing spawned task continues to do work that's helpful to the method caller. We explore Java 8 and 9 techniques for benefiting from such methods later in this chapter, starting in section 15.2, but first, check the dangers:

- The ongoing thread runs concurrently with the code following the method call and therefore requires careful programming to avoid data races.
- What happens if the Java `main()` method returns before the ongoing thread has terminated? There are two answers, both rather unsatisfactory:
  - Wait for all such outstanding threads before exiting the application.
  - Kill all outstanding threads and then exit.

www.itbook.store/books/9781617293566

The former solution risks a seeming application crash by never terminating due to a forgotten thread; the latter risks interrupting a sequence of I/O operations writing to disk, thereby leaving an external data in an inconsistent state. To avoid both of these problems, ensure that your program keeps track of all threads it creates and joins them all before exiting (including shutting down any thread pools).

Java threads can be labeled as *daemon*[4] or nondaemon, using the `setDaemon()` method call. Daemon threads are killed on exit (and therefore are useful for services that don't leave the disk in an inconsistent state), whereas returning from `main` continues to wait for all threads that aren't daemons to terminate before exiting the program.

15.1.4 What do you want from threads?

What you want is to be able to structure your program so that whenever it can benefit from parallelism, enough tasks are available to occupy all the hardware threads, which means structuring your program to have many smaller tasks (but not too small because of the cost of task switching). You saw how to do this for loops and divide-conquer algorithms in chapter 7, using parallel stream processing and fork/join, but in the rest of this chapter (and in chapters 16 and 17), you see how to do it for method calls without writing swaths of boilerplate thread-manipulation code.

15.2 Synchronous and asynchronous APIs

Chapter 7 showed you that Java 8 Streams give you a way to exploit parallel hardware. This exploitation happens in two stages. First, you replace external iteration (explicit `for` loops) with internal iteration (using Stream methods). Then you can use the `parallel()` method on Streams to allow the elements to be processed in parallel by the Java runtime library instead of rewriting every loop to use complex thread-creation operations. An additional advantage is that the runtime system is much better informed about the number of available threads when the loop is executed than is the programmer, who can only guess.

Situations other than loop-based computations can also benefit from parallelism. An important Java development that forms the background of this chapter and chapters 16 and 17 is asynchronous APIs.

Let's take for a running example the problem of summing the results of calls to methods `f` and `g` with signatures:

```
int f(int x);
int g(int x);
```

For emphasis, we'll refer to these signatures as a *synchronous API*, as they return their results when they physically return, in a sense that will soon become clear. You might

[4] Etymologically, *daemon* and *demon* arise from the same Greek word, but *daemon* captures the idea of a helpful spirit, whereas *demon* captures the idea of an evil spirit. UNIX coined the word *daemon* for computing purposes, using it for system services such as sshd, a process or thread that listens for incoming ssh connections.

www.itbook.store/books/9781617293566

invoke this API with a code fragment that calls them both and prints the sum of their results:

```
int y = f(x);
int z = g(x);
System.out.println(y + z);
```

Now suppose that methods `f` and `g` execute for a long time. (These methods could implement a mathematical optimization task, such as gradient descent, but in chapters 16 and 17, we consider more-practical cases in which they make Internet queries.) In general, the Java compiler can do nothing to optimize this code because `f` and `g` may interact in ways that aren't clear to the compiler. But if you know that `f` and `g` don't interact, or you don't care, you want to execute `f` and `g` in separate CPU cores, which makes the total execution time only the maximum of that of the calls to `f` and `g` instead of the sum. All you need to do is run the calls to `f` and `g` in separate threads. This idea is a great one, but it complicates[5] the simple code from before:

```
class ThreadExample {

    public static void main(String[] args) throws InterruptedException {
        int x = 1337;
        Result result = new Result();

        Thread t1 = new Thread(() -> { result.left = f(x); } );
        Thread t2 = new Thread(() -> { result.right = g(x); });
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(result.left + result.right);
    }

    private static class Result {
        private int left;
        private int right;
    }
}
```

You can simplify this code somewhat by using the `Future` API interface instead of `Runnable`. Assuming that you previously set up a thread pool as an `ExecutorService` (such as `executorService`), you can write

```
public class ExecutorServiceExample {
    public static void main(String[] args)
        throws ExecutionException, InterruptedException {

        int x = 1337;
```

[5] Some of the complexity here has to do with transferring results back from the thread. Only final outer-object variables can be used in lambdas or inner classes, but the real problem is all the explicit thread manipulation.

www.itbook.store/books/9781617293566

```
        ExecutorService executorService = Executors.newFixedThreadPool(2);
        Future<Integer> y = executorService.submit(() -> f(x));
        Future<Integer> z = executorService.submit(() -> g(x));
        System.out.println(y.get() + z.get());

        executorService.shutdown();
    }
}
```

but this code is still polluted by the boilerplate code involving explicit calls to `submit`.

You need a better way of expressing this idea, analogous to how internal iteration on Streams avoided the need to use thread-creation syntax to parallelize external iteration.

The answer involves changing the API to an *asynchronous API*.[6] Instead of allowing a method to return its result at the same time that it physically returns to the caller (synchronously), you allow it to return physically before producing its result, as shown in figure 15.6. Thus, the call to `f` and the code following this call (here, the call to `g`) can execute in parallel. You can achieve this parallelism by using two techniques, both of which change the signatures of `f` *and* `g`.

The first technique uses Java Futures in a better way. Futures appeared in Java 5 and were enriched into `CompletableFuture` in Java 8 to make them composable; we explain this concept in section 15.4 and explore the Java API in detail with a worked Java code example in chapter 16. The second technique is a reactive-programming style that uses the Java 9 `java.util.concurrent.Flow` interfaces, based on the publish-subscribe protocol explained in section 15.5 and exemplified with practical code in chapter 17.

How do these alternatives affect the signatures of `f` and `g`?

15.2.1 Future-style API

In this alternative, change the signature of `f` and `g` to

```
Future<Integer> f(int x);
Future<Integer> g(int x);
```

and change the calls to

```
Future<Integer> y = f(x);
Future<Integer> z = g(x);
System.out.println(y.get() + z.get());
```

The idea is that method `f` returns a Future, which contains a task that continues to evaluate its original body, but the return from `f` happens as quickly as possible after the call. Method `g` similarly returns a future, and the third code line uses `get()` to wait for both Futures to complete and sums their results.

[6] Synchronous APIs are also known as *blocking APIs,* as the physical return is delayed until the result is ready (clearest when considering a call to an I/O operation), whereas asynchronous APIs can naturally implement nonblocking I/O (where the API call merely initiates the I/O operation without waiting for the result, provided that the library at hand, such as Netty, supports nonblocking I/O operations).

www.itbook.store/books/9781617293566

In this case, you could have left the API and call of `g` unchanged without reducing parallelism—only introducing Futures for `f`. You have two reasons not to do so in bigger programs:

- Other uses of `g` may require a Future-style version, so you prefer a uniform API style.
- To enable parallel hardware to execute your programs as fast as possible, it's useful to have more and smaller tasks (within reason).

15.2.2 Reactive-style API

In the second alternative, the core idea is to use callback-style programming by changing the signature of `f` and `g` to

```
void f(int x, IntConsumer dealWithResult);
```

This alternative may seem to be surprising at first. How can `f` work if it doesn't return a value? The answer is that you instead pass a *callback*[7] (a lambda) to `f` as an additional argument, and the body of `f` spawns a task that calls this lambda with the result when it's ready instead of returning a value with `return`. Again, `f` returns immediately after spawning the task to evaluate the body, which results in the following style of code:

```
public class CallbackStyleExample {
    public static void main(String[] args) {

        int x = 1337;
        Result result = new Result();

        f(x, (int y) -> {
            result.left = y;
            System.out.println((result.left + result.right));
        } );

        g(x, (int z) -> {
            result.right = z;
            System.out.println((result.left + result.right));
        });

    }
}
```

Ah, but this isn't the same! Before this code prints the correct result (the sum of the calls to `f` and `g`), it prints the fastest value to complete (and occasionally instead prints

[7] Some authors use the term *callback* to mean any lambda or method reference passed as an argument to a method, such as the argument to `Stream.filter` or `Stream.map`. We use it only for those lambda and method references that can be called *after* the method has returned.

www.itbook.store/books/9781617293566

the sum twice, as there's no locking here, and both operands to + could be updated before either of the `println` calls is executed). There are two answers:

- You could recover the original behavior by invoking `println` after testing with if-then-else that both callbacks have been called, perhaps by counting them with appropriate locking.
- This reactive-style API is intended to react to a sequence of events, not to single results, for which `Future`s are more appropriate.

Note that this reactive style of programming allows methods `f` and `g` to invoke their callback `dealWithResult` multiple times. The original versions of `f` and `g` were obliged to use a `return` that can be performed only once. Similarly, a `Future` can be completed only once, and its result is available to `get()`. In a sense, the reactive-style asynchronous API naturally enables a sequence (which we will later liken to a stream) of values, whereas the `Future`-style API corresponds to a one-shot conceptual framework.

In section 15.5, we refine this core-idea example to model a spreadsheet call containing a formula such as `=C1+C2`.

You may argue that both alternatives make the code more complex. To some extent, this argument is correct; you shouldn't thoughtlessly use either API for every method. But APIs keep code simpler (and use higher-level constructs) than explicit thread manipulation does. Also, careful use of these APIs for method calls that (a) cause long-running computations (perhaps longer than several milliseconds) or (b) wait for a network or for input from a human can significantly improve the efficiency of your application. In case (a), these techniques make your program faster without the explicit ubiquitous use of threads polluting your program. In case (b), there's the additional benefit that the underlying system can use threads effectively without clogging up. We turn to the latter point in the next section.

15.2.3 Sleeping (and other blocking operations) considered harmful

When you're interacting with a human or an application that needs to restrict the rate at which things happen, one natural way to program is to use the `sleep()` method. A sleeping thread still occupies system resources, however. This situation doesn't matter if you have only a few threads, but it matters if you have many threads, most of which are sleeping. (See the discussion in section 15.2.1 and figure 15.3.)

The lesson to remember is that tasks sleeping in a thread pool consume resources by blocking other tasks from starting to run. (They can't stop tasks already allocated to a thread, as the operating system schedules these tasks.)

It's not only sleeping that can clog the available threads in a thread pool, of course. Any blocking operation can do the same thing. Blocking operations fall into two classes: waiting for another task to do something, such as invoking `get()` on a Future; and waiting for external interactions such as reads from networks, database servers, or human interface devices such as keyboards.

www.itbook.store/books/9781617293566

What can you do? One rather totalitarian answer is never to block within a task or at least to do so with a small number of exceptions in your code. (See section 15.2.4 for a reality check.) The better alternative is to break your task into two parts—before and after—and ask Java to schedule the after part only when it won't block.

Compare code A, shown as a single task

```
work1();
Thread.sleep(10000);        <-- Sleep for 10 seconds.
work2();
```

with code B:

```
public class ScheduledExecutorServiceExample {
    public static void main(String[] args) {
        ScheduledExecutorService scheduledExecutorService
            = Executors.newScheduledThreadPool(1);

        work1();
        scheduledExecutorService.schedule(
    ScheduledExecutorServiceExample::work2, 10, TimeUnit.SECONDS);

        scheduledExecutorService.shutdown();
    }

    public static void work1(){
        System.out.println("Hello from Work1!");
    }

    public static void work2(){
        System.out.println("Hello from Work2!");
    }
}
```

Schedule a separate task for work2() 10 seconds after work1() finishes.

Think of both tasks being executed within a thread pool.

Consider how code A executes. First, it's queued to execute in the thread pool, and later, it starts executing. Halfway through, however, it blocks in the call to sleep, occupying a worker thread for 10 whole seconds doing nothing. Then it executes `work2()` before terminating and releasing the worker thread. Code B, by comparison, executes `work1()` and then terminates—but only after having queued a task to do `work2()` 10 seconds later.

Code B is better, but why? Code A and code B do the same thing. The difference is that code A occupies a precious thread while it sleeps, whereas code B queues another task to execute (with a few bytes of memory and no requirement for a thread) instead of sleeping.

This effect is something that you should always bear in mind when creating tasks. Tasks occupy valuable resources when they start executing, so you should aim to keep them running until they complete and release their resources. Instead of blocking, a task should terminate after submitting a follow-up task to complete the work it intended to do.

www.itbook.store/books/9781617293566

Whenever possible, this guideline applies to I/O, too. Instead of doing a classical blocking read, a task should issue a nonblocking "start a read" method call and terminate after asking the runtime library to schedule a follow-up task when the read is complete.

This design pattern may seem to lead to lots of hard-to-read code. But the Java `CompletableFuture` interface (section 15.4 and chapter 16) abstracts this style of code within the runtime library, using combinators instead of explicit uses of blocking `get()` operations on `Futures`, as we discussed earlier.

As a final remark, we'll note that code A and code B would be equally effective if threads were unlimited and cheap. But they aren't, so code B is the way to go whenever you have more than a few tasks that might sleep or otherwise block.

15.2.4 Reality check

If you're designing a new system, designing it with many small, concurrent tasks so that all possible blocking operations are implemented with asynchronous calls is probably the way to go if you want to exploit parallel hardware. But reality needs to intrude into this "everything asynchronous" design principle. (Remember, "the best is the enemy of the good.") Java has had nonblocking IO primitives (`java.nio`) since Java 1.4 in 2002, and they're relatively complicated and not well known. Pragmatically, we suggest that you try to identify situations that would benefit from Java's enhanced concurrency APIs, and use them without worrying about making every API asynchronous.

You may also find it useful to look at newer libraries such as Netty (https://netty.io/), which provides a uniform blocking/nonblocking API for network servers.

15.2.5 How do exceptions work with asynchronous APIs?

In both Future-based and reactive-style asynchronous APIs, the conceptual body of the called method executes in a separate thread, and the caller's execution is likely to have exited the scope of any exception handler placed around the call. It's clear that unusual behavior that would have triggered an exception needs to perform an alternative action. But what might this action be? In the `CompletableFuture` implementation of Futures, the API includes provision for exposing exceptions at the time of the `get()` method and also provides methods such as `exceptionally()` to recover from exceptions, which we discuss in chapter 16.

For reactive-style asynchronous APIs, you have to modify the interface by introducing an additional callback, which is called instead of an exception being raised, as the existing callback is called instead of a `return` being executed. To do this, include multiple callbacks in the reactive API, as in this example:

```
void f(int x, Consumer<Integer> dealWithResult,
              Consumer<Throwable> dealWithException);
```

Then the body of `f` might perform

```
dealWithException(e);
```

www.itbook.store/books/9781617293566

If there are multiple callbacks, instead of supplying them separately, you can equivalently wrap them as methods in a single object. The Java 9 `Flow` API, for example, wraps these multiple callbacks within a single object (of class `Subscriber<T>` containing four methods interpreted as callbacks). Here are three of them:

```
void    onComplete()
void    onError(Throwable throwable)
void    onNext(T item)
```

Separate callbacks indicate when a value is available (`onNext`), when an exception arose while trying to make a value available (`onError`), and when an `onComplete` callback enables the program to indicate that no further values (or exceptions) will be produced. For the preceding example, the API for `f` would now be

```
void f(int x, Subscriber<Integer> s);
```

and the body of `f` would now indicate an exception, represented as `Throwable t`, by performing

```
s.onError(t);
```

Compare this API containing multiple callbacks with reading numbers from a file or keyboard device. If you think of such a device as being a producer rather than a passive data structure, it produces a sequence of "Here's a number" or "Here's a malformed item instead of a number" items, and finally a "There are no more characters left (end-of-file)" notification.

It's common to refer to these calls as messages, or *events*. You might say, for example, that the file reader produced the number events 3, 7, and 42, followed by a malformed-number event, followed by the number event 2 and then by the end-of-file event.

When seeing these events as part of an API, it's important to note that the API signifies nothing about the relative ordering of these events (often called the *channel protocol*). In practice, the accompanying documentation specifies the protocol by using phases such as "After an `onComplete` event, no more events will be produced."

15.3 The box-and-channel model

Often, the best way to design and think about concurrent systems is pictorially. We call this technique the *box-and-channel model*. Consider a simple situation involving integers, generalizing the earlier example of calculating `f(x) + g(x)`. Now you want to call method or function `p` with argument `x`, pass its result to functions `q1` and `q2`, call method or function `r` with the results of these two calls, and then print the result. (To avoid clutter in this explanation, we're not going to distinguish between a method `m` of class `C` and its associated function `C::m`.) Pictorially, this task is simple, as shown in figure 15.7.

www.itbook.store/books/9781617293566

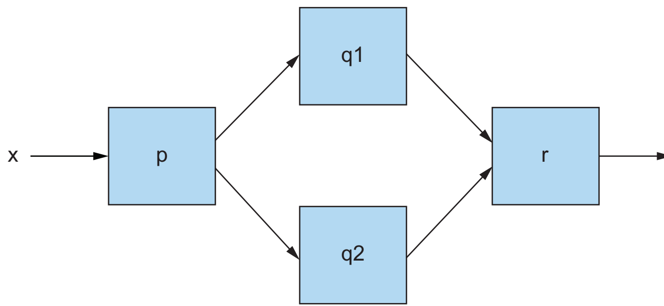

Figure 15.7 A simple box-and-channel diagram

Look at two ways of coding figure 15.7 in Java to see the problems they cause. The first way is

```
int t = p(x);
System.out.println( r(q1(t), q2(t)) );
```

This code appears to be clear, but Java runs the calls to `q1` and `q2` in turn, which is what you want to avoid when trying to exploit hardware parallelism.

Another way is to use Futures to evaluate `f` and `g` in parallel:

```
int t = p(x);
Future<Integer> a1 = executorService.submit(() -> q1(t));
Future<Integer> a2 = executorService.submit(() -> q2(t));
System.out.println( r(a1.get(),a2.get()));
```

Note: We didn't wrap `p` and `r` in Futures in this example because of the shape of the box-and-channel diagram. `p` has to be done before everything else and `r` after everything else. This would no longer be the case if we changed the example to mimic

```
System.out.println( r(q1(t), q2(t)) + s(x) );
```

in which we'd need to wrap all five functions (`p`, `q1`, `q2`, `r`, and `s`) in Futures to maximize concurrency.

This solution works well if the total amount of concurrency in the system is small. But what if the system becomes large, with many separate box-and-channel diagrams, and with some of the boxes themselves internally using their own boxes and channels? In this situation, many tasks might be waiting (with a call to `get()`) for a Future to complete, and as discussed in section 15.1.2, the result may be underexploitation of hardware parallelism or even deadlock. Moreover, it tends to be hard to understand such large-scale system structure well enough to work out how many tasks are liable to be waiting for a `get()`. The solution that Java 8 adopts (`CompletableFuture`; see section 15.4 for details) is to use *combinators*. You've already seen that you can use methods such as `compose()` and `andThen()` on two `Functions` to get another `Function` (see chapter 3). Assuming that `add1` adds 1 to an Integer and that `dble` doubles an Integer, for example, you can write

```
Function<Integer, Integer> myfun = add1.andThen(dble);
```

www.itbook.store/books/9781617293566

to create a `Function` that doubles its argument and adds 2 to the result. But box-and-channel diagrams can be also coded directly and nicely with combinators. Figure 15.7 could be captured succinctly with Java `Functions` `p, q1, q2` and `BiFunction` `r` as

```
p.thenBoth(q1,q2).thenCombine(r)
```

Unfortunately, neither `thenBoth` nor `thenCombine` is part of the Java `Function` and `BiFunction` classes in exactly this form.

In the next section, you see how similar ideas of combinators work for `CompletableFuture` and prevent tasks from ever to have to wait using `get()`.

Before leaving this section, we want to emphasize the fact that the box-and-channel model can be used to structure thoughts and code. In an important sense, it raises the level of abstraction for constructing a larger system. You draw boxes (or use combinators in programs) to express the computation you want, which is later executed, perhaps more efficiently than you might have obtained by hand-coding the computation. This use of combinators works not only for mathematical functions, but also for Futures and reactive streams of data. In section 15.5, we generalize these box-and-channel diagrams into marble diagrams in which multiple marbles (representing messages) are shown on every channel. The box-and-channel model also helps you change perspective from directly programming concurrency to allowing combinators to do the work internally. Similarly, Java 8 Streams change perspective from the coder having to iterate over a data structure to combinators on Streams doing the work internally.

15.4 CompletableFuture and combinators for concurrency

One problem with the `Future` interface is that it's an interface, encouraging you to think of and structure your concurrent coding tasks as Futures. Historically, however, Futures have provided few actions beyond `FutureTask` implementations: creating a future with a given computation, running it, waiting for it to terminate, and so on. Later versions of Java provided more structured support (such as `RecursiveTask`, discussed in chapter 7).

What Java 8 brings to the party is the ability to compose Futures, using the `CompletableFuture` implementation of the `Future` interface. So why call it `CompletableFuture` rather than, say, `ComposableFuture`? Well, an ordinary Future is typically created with a `Callable`, which is run, and the result is obtained with a `get()`. But a `CompletableFuture` allows you to create a Future without giving it any code to run, and a `complete()` method allows some other thread to complete it later with a value (hence the name) so that `get()` can access that value. To sum `f(x)` and `g(x)` concurrently, you can write

```
public class CFComplete {

    public static void main(String[] args)
        throws ExecutionException, InterruptedException {
```

www.itbook.store/books/9781617293566

```
        ExecutorService executorService = Executors.newFixedThreadPool(10);
        int x = 1337;

        CompletableFuture<Integer> a = new CompletableFuture<>();
        executorService.submit(() -> a.complete(f(x)));
        int b = g(x);
        System.out.println(a.get() + b);

        executorService.shutdown();
    }
}
```

or you can write

```
public class CFComplete {

    public static void main(String[] args)
        throws ExecutionException, InterruptedException {
        ExecutorService executorService = Executors.newFixedThreadPool(10);
        int x = 1337;

        CompletableFuture<Integer> a = new CompletableFuture<>();
        executorService.submit(() -> b.complete(g(x)));
        int a = f(x);
        System.out.println(a + b.get());

        executorService.shutdown();
    }
}
```

Note that both these code versions can waste processing resources (recall section 15.2.3) by having a thread blocked waiting for a `get`—the former if `f(x)` takes longer, and the latter if `g(x)` takes longer. Using Java 8's `CompletableFuture` enables you to avoid this situation; but first a quiz.

Quiz 15.1:

Before reading further, think how you might write tasks to exploit threads perfectly in this case: two active threads while both `f(x)` and `g(x)` are executing, and one thread starting from when the first one completes up to the return statement.

The answer is that you'd use one task to execute `f(x)`, a second task to execute `g(x)`, and a third task (a new one or one of the existing ones) to calculate the sum, and somehow, the third task can't start before the first two finish. How do you solve this problem in Java?

The solution is to use the idea of composition on Futures.

First, refresh your memory about composing operations, which you've seen twice before in this book. Composing operations is a powerful program-structuring idea used in many other languages, but it took off in Java only with the addition of lambdas

www.itbook.store/books/9781617293566

in Java 8. One instance of this idea of composition is composing operations on streams, as in this example:

```
myStream.map(…).filter(…).sum()
```

Another instance of this idea is using methods such as `compose()` and `andThen()` on two `Function`s to get another `Function` (see section 15.5).

This gives you a new and better way to add the results of your two computations by using the `thenCombine` method from `CompletableFuture<T>`. Don't worry too much about the details at the moment; we discuss this topic more comprehensively in chapter 16. The method `thenCombine` has the following signature (slightly simplified to prevent the clutter associated with generics and wildcards):

```
CompletableFuture<V> thenCombine(CompletableFuture<U> other,
                                 BiFunction<T, U, V> fn)
```

The method takes two `CompletableFuture` values (with result types `T` and `U`) and creates a new one (with result type `V`). When the first two complete, it takes both their results, applies `fn` to both results, and completes the resulting future without blocking. The preceding code could now be rewritten in the following form:

```
public class CFCombine {

    public static void main(String[] args) throws ExecutionException,
     InterruptedException {

        ExecutorService executorService = Executors.newFixedThreadPool(10);
        int x = 1337;

        CompletableFuture<Integer> a = new CompletableFuture<>();
        CompletableFuture<Integer> b = new CompletableFuture<>();
        CompletableFuture<Integer> c = a.thenCombine(b, (y, z)-> y + z);
        executorService.submit(() -> a.complete(f(x)));
        executorService.submit(() -> b.complete(g(x)));


        System.out.println(c.get());
        executorService.shutdown();

    }
}
```

The `thenCombine` line is critical: without knowing anything about computations in the Futures `a` and `b`, it creates a computation that's scheduled to run in the thread pool only when both of the first two computations have completed. The third computation, `c`, adds their results and (most important) isn't considered to be eligible to execute on a thread until the other two computations have completed (rather than starting to execute early and then blocking). Therefore, no actual wait operation is performed,

www.itbook.store/books/9781617293566

which was troublesome in the earlier two versions of this code. In those versions, if the computation in the Future happens to finish second, two threads in the thread pool are still active, even though you need only one! Figure 15.8 shows this situation diagrammatically. In both earlier versions, calculating `y+z` happens on the same fixed thread that calculates `f(x)` or `g(x)`—with a potential wait in between. By contrast, using `thenCombine` schedules the summing computation only after both `f(x)` and `g(x)` have completed.

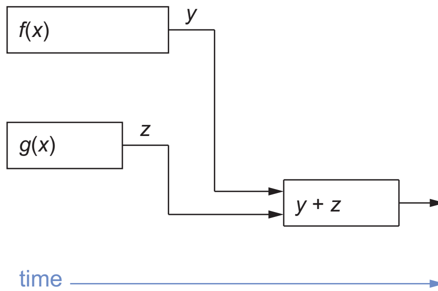

Figure 15.8 Timing diagram showing three computations: `f(x)`, `g(x)` and adding their results

To be clear, for many pieces of code, you don't need to worry about a few threads being blocked waiting for a `get()`, so pre-Java 8 Futures remain sensible programming options. In some situations, however, you want to have a large number of Futures (such as for dealing with multiple queries to services). In these cases, using `CompletableFuture` and its combinators to avoid blocking calls to `get()` and possible loss of parallelism or deadlock is often the best solution.

15.5 Publish-subscribe and reactive programming

The mental model for a `Future` and `CompletableFuture` is that of a computation that executes independently and concurrently. The result of the Future is available with `get()` after the computation completes. Thus, Futures are *one-shot*, executing code that runs to completion only once.

By contrast, the mental model for reactive programming is a Future-like object that, over time, yields multiple results. Consider two examples, starting with a thermometer object. You expect this object to yield a result repeatedly, giving you a temperature value every few seconds. Another example is an object representing the listener component of a web server; this object waits until an HTTP request appears over the network and similarly yields with the data from the request. Then other code can process the result: a temperature or data from an HTTP request. Then the thermometer and listener objects go back to sensing temperatures or listening before potentially yielding further results.

Note two points here. The core point is that these examples are like Futures but differ in that they can complete (or yield) multiple times instead of being one-shot. Another point is that in the second example, earlier results may be as important as ones seen later, whereas for a thermometer, most users are interested only in the

www.itbook.store/books/9781617293566

most-recent temperature. But why is this type of a programming called *reactive*? The answer is that another part of the program may want to *react* to a low temperature report (such as by turning on a heater).

You may think that the preceding idea is only a Stream. If your program fits naturally into the Stream model, a Stream may be the best implementation. In general, though, the reactive-programming paradigm is more expressive. A given Java Stream can be consumed by only one terminal operation. As we mention in section 15.3, the Stream paradigm makes it hard to express Stream-like operations that can split a sequence of values between two processing pipelines (think fork) or process and combine items from two separate streams (think join). Streams have linear processing pipelines.

Java 9 models reactive programming with interfaces available inside `java.util.concurrent.Flow` and encodes what's known as the publish-subscribe model (or protocol, often shortened to pub-sub). You learn about the Java 9 Flow API in more detail in chapter 17, but we provide a short overview here. There are three main concepts:

- A *publisher* to which a *subscriber* can subscribe.
- The connection is known as a *subscription*.
- *Messages* (also known an *events*) are transmitted via the connection.

Figure 15.9 shows the idea pictorially, with subscriptions as channels and publishers and subscribers as ports on boxes. Multiple components can subscribe to a single publisher, a component can publish multiple separate streams, and a component can subscribe to multiple publishers. In this next section, we show you how this idea works step by step, using the nomenclature of the Java 9 Flow interface.

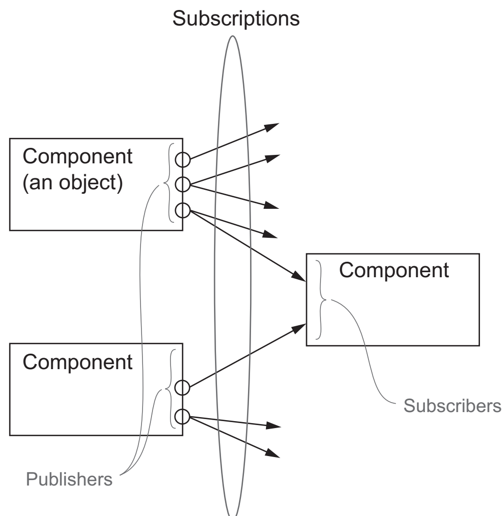

Figure 15.9 The publish-subscribe model

www.itbook.store/books/9781617293566

15.5.1 Example use for summing two flows

A simple but characteristic example of publish-subscribe combines events from two sources of information and publishes them for others to see. This process may sound obscure at first, but it's what a cell containing a formula in a spreadsheet does conceptually. Model a spreadsheet cell C3, which contains the formula `"=C1+C2"`. Whenever cell C1 or C2 is updated (by a human or because the cell contains a further formula), C3 is updated to reflect the change. The following code assumes that the only operation available is adding the values of cells.

First, model the concept of a cell that holds a value:

```
private class SimpleCell {
    private int value = 0;
    private String name;

    public SimpleCell(String name) {
        this.name = name;
    }
}
```

At the moment, the code is simple, and you can initialize a few cells, as follows:

```
SimpleCell c2 = new SimpleCell("C2");
SimpleCell c1 = new SimpleCell("C1");
```

How do you specify that when the value of `c1` or `c2` changes, `c3` sums the two values? You need a way for `c1` and `c2` to subscribe `c3` to their events. To do so, introduce the interface `Publisher<T>`, which at its core looks like this:

```
interface Publisher<T> {
    void subscribe(Subscriber<? super T> subscriber);
}
```

This interface takes a subscriber as an argument that it can communicate with. The `Subscriber<T>` interface includes a simple method, `onNext`, that takes that information as an argument and then is free to provide a specific implementation:

```
interface Subscriber<T> {
    void onNext(T t);
}
```

How do you bring these two concepts together? You may realize that a `Cell` is in fact both a `Publisher` (can subscribe cells to its events) and a `Subscriber` (reacts to events from other cells). The implementation of the `Cell` class now looks like this:

```
private class SimpleCell implements Publisher<Integer>, Subscriber<Integer> {
    private int value = 0;
    private String name;
    private List<Subscriber> subscribers = new ArrayList<>();
```

www.itbook.store/books/9781617293566

```
    public SimpleCell(String name) {
        this.name = name;
    }

    @Override
    public void subscribe(Subscriber<? super Integer> subscriber) {
        subscribers.add(subscriber);
    }

    private void notifyAllSubscribers() {
        subscribers.forEach(subscriber -> subscriber.onNext(this.value));
    }

    @Override
    public void onNext(Integer newValue) {
        this.value = newValue;
        System.out.println(this.name + ":" + this.value);
        notifyAllSubscribers();
    }
}
```

Reacts to a new value from a cell it is subscribed to by updating its value

This method notifies all the subscribers with a new value.

Notifies all subscribers about the updated value

Prints the value in the console but could be rendering the updated cell as part of an UI

Try a simple example:

```
Simplecell c3 = new SimpleCell("C3");
SimpleCell c2 = new SimpleCell("C2");
SimpleCell c1 = new SimpleCell("C1");

c1.subscribe(c3);

c1.onNext(10); // Update value of C1 to 10
c2.onNext(20); // update value of C2 to 20
```

This code outputs the following result because C3 is directly subscribed to C1:

```
C1:10
C3:10
C2:20
```

How do you implement the behavior of `"C3=C1+C2"` ? You need to introduce a separate class that's capable of storing two sides of an arithmetic operation (left and right):

```
    public class ArithmeticCell extends SimpleCell {

        private int left;
        private int right;

        public ArithmeticCell(String name) {
            super(name);
        }

        public void setLeft(int left) {
            this.left = left;
            onNext(left + this.right);
        }
```

Update the cell value and notify any subscribers.

www.itbook.store/books/9781617293566

```
        public void setRight(int right) {
            this.right = right;
            onNext(right + this.left);
        }
    }
```

Update the cell value and notify any subscribers.

Now you can try a more-realistic example:

```
ArithmeticCell c3 = new ArithmeticCell("C3");
SimpleCell c2 = new SimpleCell("C2");
SimpleCell c1 = new SimpleCell("C1");

c1.subscribe(c3::setLeft);
c2.subscribe(c3::setRight);

c1.onNext(10); // Update value of C1 to 10
c2.onNext(20); // update value of C2 to 20
c1.onNext(15); // update value of C1 to 15
```

The output is

```
C1:10
C3:10
C2:20
C3:30
C1:15
C3:35
```

By inspecting the output, you see that when C1 was updated to 15, C3 immediately reacted and updated its value as well. What's neat about the publisher-subscriber interaction is the fact that you can set up a graph of publishers and subscribers. You could create another cell C5 that depends on C3 and C4 by expressing `"C5=C3+C4"`, for example:

```
ArithmeticCell c5 = new ArithmeticCell("C5");
ArithmeticCell c3 = new ArithmeticCell("C3");
SimpleCell c4 = new SimpleCell("C4");
SimpleCell c2 = new SimpleCell("C2");
SimpleCell c1 = new SimpleCell("C1");


c1.subscribe(c3::setLeft);
c2.subscribe(c3::setRight);

c3.subscribe(c5::setLeft);
c4.subscribe(c5::setRight);
```

Then you can perform various updates in your spreadsheet:

```
c1.onNext(10); // Update value of C1 to 10
c2.onNext(20); // update value of C2 to 20
c1.onNext(15); // update value of C1 to 15
c4.onNext(1); // update value of C4 to 1
c4.onNext(3); // update value of C4 to 3
```

www.itbook.store/books/9781617293566

These actions result in the following output:

```
C1:10
C3:10
C5:10
C2:20
C3:30
C5:30
C1:15
C3:35
C5:35
C4:1
C5:36
C4:3
C5:38
```

In the end, the value of `C5` is `38` because `C1` is `15`, `C2` is `20`, and `C4` is `3`.

Nomenclature

Because data flows from publisher (producer) to subscriber (consumer), developers often use words such as *upstream* and *downstream*. In the preceding code examples, the data `newValue` received by the upstream `onNext()` methods is passed via the call to `notifyAllSubscribers()` to the downstream `onNext()` call.

That's the core idea of publish-subscribe. We've left out a few things, however, some of which are straightforward embellishments, and one of which (backpressure) is so vital that we discuss it separately in the next section.

First, we'll discuss the straightforward things. As we remark in section 15.2, practical programming of flows may want to signal things other than an `onNext` event, so subscribers (listeners) need to define `onError` and `onComplete` methods so that the publisher can indicate exceptions and terminations of data flow. (Perhaps the example of a thermometer has been replaced and will never produce more values via `onNext`.) The methods `onError` and `onComplete` are supported in the actual `Subscriber` interface in the Java 9 Flow API. These methods are among the reasons why this protocol is more powerful than the traditional Observer pattern.

Two simple but vital ideas that significantly complicate the Flow interfaces are pressure and backpressure. These ideas can appear to be unimportant, but they're vital for thread utilization. Suppose that your thermometer, which previously reported a temperature every few seconds, was upgraded to a better one that reports a temperature every millisecond. Could your program react to these events sufficiently quickly, or might some buffer overflow and cause a crash? (Recall the problems giving thread pools large numbers of tasks if more than a few tasks might block.) Similarly, suppose that you subscribe to a publisher that furnishes all the SMS messages onto your phone. The subscription might work well on my newish phone with only a few SMS messages, but what happens in a few years when there are thousands of messages, all

www.itbook.store/books/9781617293566

potentially sent via calls to onNext in less than a second? This situation is often known as *pressure*.

Now think of a vertical pipe containing messages written on balls. You also need a form of backpressure, such as a mechanism that restricts the number of balls being added to the column. Backpressure is implemented in the Java 9 Flow API by a request() method (in a new interface called Subscription) that invites the publisher to send the next item(s), instead of items being sent at an unlimited rate (the pull model instead of the push model). We turn this topic in the next section.

15.5.2 Backpressure

You've seen how to pass a Subscriber object (containing onNext, onError, and OnComplete methods) to a Publisher, which the publisher calls when appropriate. This object passes information from Publisher to Subscriber. You want to limit the rate at which this information is sent via backpressure (flow control), which requires you to send information from Subscriber to Publisher. The problem is that the Publisher may have multiple Subscribers, and you want backpressure to affect only the point-to-point connection involved. In the Java 9 Flow API, the Subscriber interface includes a fourth method

```
void onSubscribe(Subscription subscription);
```

that's called as the first event sent on the channel established between Publisher and Subscriber. The Subscription object contains methods that enable the Subscriber to communicate with the Publisher, as follows:

```
interface Subscription {
    void   cancel();
    void   request(long n);
}
```

Note the usual "this seems backward" effect with callbacks. The Publisher creates the Subscription object and passes it to the Subscriber, which can call its methods to pass information from the Subscriber back to the Publisher.

15.5.3 A simple form of real backpressure

To enable a publish-subscribe connection to deal with events one at a time, you need to make the following changes:

- Arrange for the Subscriber to store the Subscription object passed by OnSubscribe locally, perhaps as a field subscription.
- Make the last action of onSubscribe, onNext, and (perhaps) onError be a call to channel.request(1) to request the next event (only one event, which stops the Subscriber from being overwhelmed).
- Change the Publisher so that notifyAllSubscribers (in this example) sends an onNext or onError event along only the channels that made a request.

www.itbook.store/books/9781617293566

(Typically, the `Publisher` creates a new `Subscription` object to associate with each `Subscriber` so that multiple `Subscribers` can each process data at their own rate.)

Although this process seems to be simple, implementing backpressure requires thinking about a range of implementation trade-offs:

- Do you send events to multiple `Subscribers` at the speed of the slowest, or do you have a separate queue of as-yet-unsent data for each `Subscriber`?
- What happens when these queues grow excessively?
- Do you drop events if the `Subscriber` isn't ready for them?

The choice depends on the semantics of the data being sent. Losing one temperature report from a sequence may not matter, but losing a credit in your bank account certainly does!

You often hear this concept referred to as reactive pull-based backpressure. The concept is called *reactive pull*-based because it provides a way for the `Subscriber` to pull (request) more information from the `Publisher` via events (reactive). The result is a backpressure mechanism.

15.6 Reactive systems vs. reactive programming

Increasingly in the programming and academic communities, you may hear about reactive systems and reactive programming, and it's important to realize that these terms express quite different ideas.

A *reactive system* is a program whose architecture allows it to react to changes in its runtime environments. Properties that reactive systems should have are formalized in the Reactive Manifesto (http://www.reactivemanifesto.org) (see chapter 17). Three of these properties can be summarized as responsive, resilient, and elastic.

Responsive means that a reactive system can respond to inputs in real time rather delaying a simple query because the system is processing a big job for someone else. *Resilient* means that a system generally doesn't fail because one component fails; a broken network link shouldn't affect queries that don't involve that link, and queries to an unresponsive component can be rerouted to an alternative component. *Elastic* means that a system can adjust to changes in its workload and continue to execute efficiently. As you might dynamically reallocate staff in a bar between serving food and serving drinks so that wait times in both lines are similar, you might adjust the number of worker threads associated with various software services so that no worker is idle while ensuring that each queue continues to be processed.

Clearly, you can achieve these properties in many ways, but one main approach is to use *reactive programming* style, provided in Java by interfaces associated with `java.util.concurrent.Flow`. The design of these interfaces reflects the fourth and final property of the Reactive Manifesto: being message-driven. *Message-driven* systems have internal APIs based on the box-and-channel model, with components waiting for

www.itbook.store/books/9781617293566

inputs that are processed, with the results sent as messages to other components to enable the system to be responsive.

15.7 Road map

Chapter 16 explores the `CompletableFuture` API with a real Java example, and chapter 17 explores the Java 9 Flow (publish-subscribe) API.

Summary

- Support for concurrency in Java has evolved and continues to evolve. Thread pools are generally helpful but can cause problems when you have many tasks that can block.
- Making methods asynchronous (returning before all their work is done) allows additional parallelism, complementary to that used to optimize loops.
- You can use the box-and-channel model to visualize asynchronous systems.
- The Java 8 `CompletableFuture` class and the Java 9 Flow API can both represent box-and-channel diagrams.
- The `CompletableFuture` class expresses one-shot asynchronous computations. Combinators can be used to compose asynchronous computations without the risk of blocking that's inherent in traditional uses of Futures.
- The Flow API is based on the publish-subscribe protocol, including backpressure, and forms the basis for reactive programming in Java.
- Reactive programming can be used to implement a reactive system.

www.itbook.store/books/9781617293566

JAVA

Modern Java IN ACTION

Urma • Fusco • Mycroft

Modern applications take advantage of innovative designs, including microservices, reactive architectures, and streaming data. Modern Java features like lambdas, streams, and the long-awaited Java Module System make implementing these designs significantly easier. It's time to upgrade your skills and meet these challenges head on!

Modern Java in Action connects new features of the Java language with their practical applications. Using crystal-clear examples and careful attention to detail, this book respects your time. It will help you expand your existing knowledge of core Java as you master modern additions like the Streams API and the Java Module System, explore new approaches to concurrency, and learn how functional concepts can help you write code that's easier to read and maintain.

What's Inside

- Thoroughly revised edition of Manning's bestselling *Java 8 in Action*
- New features in Java 8, Java 9, and beyond
- Streaming data and reactive programming
- The Java Module System

Written for developers familiar with core Java features.

Raoul-Gabriel Urma is CEO of Cambridge Spark. **Mario Fusco** is a senior software engineer at Red Hat. **Alan Mycroft** is a University of Cambridge computer science professor; he cofounded the Raspberry Pi Foundation.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/modern-java-in-action

"A comprehensive and practical introduction to the modern features of the latest Java releases with excellent examples!"
—Oleksandr Mandryk
EPAM Systems

"Hands-on Java 8 and 9, simply and elegantly explained."
—Deepak Bhaskaran, Salesforce

"A lot of great examples and use cases for streams, concurrency, and reactive programming."
—Rob Pacheco, Synopsys

"My Java code improved significantly after reading this book. I was able to take the clear examples and immediately put them into practice."
—Holly Cummins, IBM

ISBN-13: 978-1-61729-356-6
ISBN-10: 1-61729-356-3

$54.99 / Can $72.99 [INCLUDING eBOOK]

www.itbook.store/books/9781617293566