16 CompletableFuture: composable asynchronous programming

This chapter covers

- Creating an asynchronous computation and retrieving its result
- Increasing throughput by using nonblocking operations
- Designing and implementing an asynchronous API
- Consuming asynchronously a synchronous API
- Pipelining and merging two or more asynchronous operations
- Reacting to the completion of an asynchronous operation

Chapter 15 explored the modern concurrency context: that multiple processing resources (CPU cores and the like) are available, and you want your programs to exploit as many of these resources as possible in a high-level manner (rather than litter your programs with ill-structured, unmaintainable operations on threads). We noted that parallel streams and fork/join parallelism provide higher-level constructs for expressing parallelism in programs iterating over collections and in programs involving divide-and-conquer, but that method invocations provide

additional opportunities for executing code in parallel. Java 8 and 9 introduce two specific APIs for this purpose: `CompletableFuture` and the reactive-programming paradigm. This chapter explains, through practical code examples, how the Java 8 `CompletableFuture` implementation of the `Future` interface gives you additional weapons in your programming armory. It also discusses additions introduced in Java 9.

16.1 Simple use of Futures

The `Future` interface was introduced in Java 5 to model a result made available at some point in the future. A query to a remote service won't be available immediately when the caller makes the request, for example. The `Future` interface models an asynchronous computation and provides a reference to its result that becomes available when the computation itself is completed. Triggering a potentially time-consuming action inside a `Future` allows the caller `Thread` to continue doing useful work instead of waiting for the operation's result. You can think of this process as being like taking a bag of clothes to your favorite dry cleaner. The cleaner gives you a receipt to tell you when your clothes will be cleaned (a `Future`); in the meantime, you can do some other activities. Another advantage of `Future` is that it's friendlier to work with than lower-level `Threads`. To work with a `Future`, you typically have to wrap the time-consuming operation inside a `Callable` object and submit it to an `ExecutorService`. The following listing shows an example written before Java 8.

Listing 16.1 Executing a long-lasting operation asynchronously in a `Future`

Submit a Callable to the ExecutorService.

Create an ExecutorService allowing you to submit tasks to a thread pool.

```
ExecutorService executor = Executors.newCachedThreadPool();
Future<Double> future = executor.submit(new Callable<Double>() {
        public Double call() {
            return doSomeLongComputation();
        }});
doSomethingElse();
try {
    Double result = future.get(1, TimeUnit.SECONDS);
} catch (ExecutionException ee) {
    // the computation threw an exception
} catch (InterruptedException ie) {
    // the current thread was interrupted while waiting
} catch (TimeoutException te) {
    // the timeout expired before the Future completion
}
```

Do something else while the asynchronous operation is progressing.

Retrieve the result of the asynchronous operation, blocking if it isn't available yet but waiting for 1 second at most before timing out.

Execute a long operation asynchronously in a separate thread.

As depicted in figure 16.1, this style of programming allows your thread to perform some other tasks while the long-lasting operation is executed concurrently in a separate thread provided by the `ExecutorService`. Then, when you can't do any other

meaningful work without having the result of that asynchronous operation, you can retrieve it from the `Future` by invoking its `get` method. This method immediately returns the result of the operation if it's already completed or blocks your thread, waiting for its result to be available.

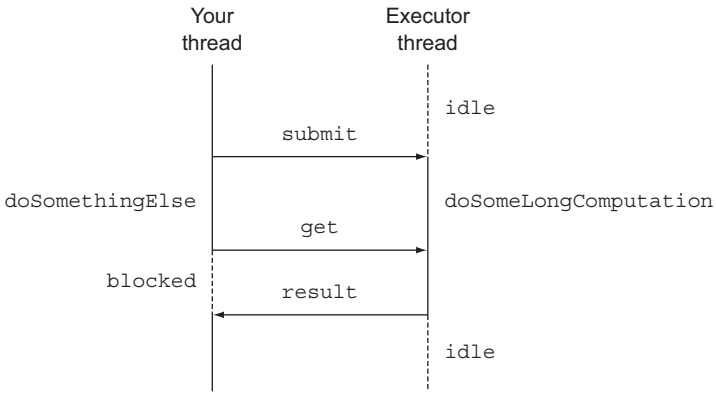

Figure 16.1 Using a `Future` to execute a long operation asynchronously

Note the problem with this scenario. What if the long operation never returns? To handle this possibility, it's almost always a good idea to use the two-argument version of `get`, which takes a timeout specifying the maximum time (along with its time unit) that your thread is willing to wait for the `Future`'s result (as in listing 16.1). The zero-argument version of `get` would instead wait indefinitely.

16.1.1 Understanding Futures and their limitations

This first small example shows that the `Future` interface provides methods for checking whether the asynchronous computation is complete (by using the `isDone` method), waiting for its completion, and retrieving its result. But these features aren't enough to let you write concise concurrent code. It's difficult, for example, to express dependencies among results of a `Future`. Declaratively, it's easy to specify, "When the result of the long computation is available, please send its result to another long computation, and when that's done, combine its result with the result from another query." Implementing this specification with the operations available in a `Future` is a different story, which is why it would be useful to have more declarative features in the implementation, such as these:

- Combining two asynchronous computations both when they're independent and when the second depends on the result of the first
- Waiting for the completion of all tasks performed by a set of `Futures`
- Waiting for the completion of only the quickest task in a set of `Futures` (possibly because the `Futures` are trying to calculate the same value in different ways) and retrieving its result

- Programmatically completing a `Future` (that is, by providing the result of the asynchronous operation manually)
- Reacting to a `Future` completion (that is, being notified when the completion happens and then being able to perform a further action with the result of the `Future` instead of being blocked while waiting for its result)

In the rest of this chapter, you learn how the `CompletableFuture` class (which implements the `Future` interface) makes all these things possible in a declarative way by means of Java 8's new features. The designs of `Stream` and `CompletableFuture` follow similar patterns, because both use lambda expressions and pipelining. For this reason, you could say that `CompletableFuture` is to a plain `Future` what `Stream` is to a `Collection`.

16.1.2 Using CompletableFutures to build an asynchronous application

To explore the `CompletableFuture` features, in this section you incrementally develop a best-price-finder application that contacts multiple online shops to find the lowest price for a given product or service. Along the way, you learn several important skills:

- How to provide an asynchronous API for your customers (useful if you're the owner of one of the online shops).
- How to make your code nonblocking when you're a consumer of a synchronous API. You discover how to pipeline two subsequent asynchronous operations, merging them into a single asynchronous computation. This situation arises, for example, when the online shop returns a discount code along with the original price of the item you wanted to buy. You have to contact a second remote discount service to find out the percentage discount associated with this discount code before calculating the actual price of that item.
- How to reactively process events representing the completion of an asynchronous operation and how doing so allows the best-price-finder application to constantly update the best-buy quote for the item you want to buy as each shop returns its price, instead of waiting for all the shops to return their respective quotes. This skill also averts the scenario in which the user sees a blank screen forever if one of the shops' servers is down.

Synchronous vs. asynchronous API

The phrase *synchronous API* is another way of talking about a traditional call to a method: you call it, the caller waits while the method computes, the method returns, and the caller continues with the returned value. Even if the caller and callee were executed on different threads, the caller would still wait for the callee to complete. This situation gives rise to the phrase *blocking call*.

By contrast, in an *asynchronous API* the method returns immediately (or at least before its computation is complete), delegating its remaining computation to a thread, which runs asynchronously to the caller—hence, the phrase *nonblocking call*. The remaining computation gives its value to the caller by calling a callback method,

or the caller invokes a further "wait until the computation is complete" method. This style of computation is common in I/O systems programming: you initiate a disc access, which happens asynchronously while you do more computation, and when you have nothing more useful to do, you wait until the disc blocks are loaded into memory. Note that blocking and nonblocking are often used for specific implementations of I/O by the operating system. However, these terms tend to be used interchangeably with asynchronous and synchronous even in non-I/O contexts.

16.2 Implementing an asynchronous API

To start implementing the best-price-finder application, define the API that each shop should provide. First, a shop declares a method that returns the price of a product, given its name:

```
public class Shop {
    public double getPrice(String product) {
        // to be implemented
    }
}
```

The internal implementation of this method would query the shop's database, but probably also perform other time-consuming tasks, such as contacting other external services (such as the shop's suppliers or manufacturer-related promotional discounts). To fake such a long-running method execution, in the rest of this chapter you use a `delay` method, which introduces an artificial delay of 1 second, as defined in the following listing.

Listing 16.2 A method to simulate a 1-second delay

```
public static void delay() {
    try {
        Thread.sleep(1000L);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
```

For the purposes of this chapter, you can model the `getPrice` method by calling `delay` and then returning a randomly calculated value for the price, as shown in the next listing. The code for returning a randomly calculated price may look like a bit of a hack; it randomizes the price based on the product name by using the result of `charAt` as a number.

Listing 16.3 Introducing a simulated delay in the `getPrice` method

```
public double getPrice(String product) {
    return calculatePrice(product);
}
```

```
private double calculatePrice(String product) {
    delay();
    return random.nextDouble() * product.charAt(0) + product.charAt(1);
}
```

This code implies that when the consumer of this API (in this case, the best-price-finder application) invokes this method, it remains blocked and then is idle for 1 second while waiting for its synchronous completion. This situation is unacceptable, especially considering that the best-price-finder application has to repeat this operation for all the shops in its network. In the subsequent sections of this chapter, you discover how to resolve this problem by consuming this synchronous API in an asynchronous way. But for the purposes of learning how to design an asynchronous API, you continue in this section to pretend to be on the other side of the barricade. You're a wise shop owner who realizes how painful this synchronous API is for its users, and you want to rewrite it as an asynchronous API to make your customers' lives easier.

16.2.1 Converting a synchronous method into an asynchronous one

To achieve this goal, you first have to turn the `getPrice` method into a `getPriceAsync` method and change its return value, as follows:

```
public Future<Double> getPriceAsync(String product) { ... }
```

As we mentioned in the introduction of this chapter, the `java.util.concurrent.Future` interface was introduced in Java 5 to represent the result of an asynchronous computation. (That is, the caller thread is allowed to proceed without blocking.) A `Future` is a handle for a value that isn't available yet but that can be retrieved by invoking its `get` method after its computation finally terminates. As a result, the `getPriceAsync` method can return immediately, giving the caller thread a chance to perform other useful computations in the meantime. The Java 8 `CompletableFuture` class gives you various possibilities to implement this method easily, as shown in the following listing.

Listing 16.4 Implementing the `getPriceAsync` method

```
public Future<Double> getPriceAsync(String product) {
    CompletableFuture<Double> futurePrice = new CompletableFuture<>();
    new Thread( () -> {
                double price = calculatePrice(product);
                futurePrice.complete(price);
    }).start();
    return futurePrice;
}
```

Execute the computation asynchronously in a different Thread.

Create the CompletableFuture that will contain the result of the computation.

Set the value returned by the long computation on the Future when it becomes available.

Return the Future without waiting for the computation of the result it contains to be completed.

Here, you create an instance of `CompletableFuture`, representing an asynchronous computation and containing a result when it becomes available. Then you fork a

different `Thread` that will perform the actual price calculation and return the `Future` instance without waiting for that long-lasting calculation to terminate. When the price of the requested product is finally available, you can complete the `CompletableFuture`, using its `complete` method to set the value. This feature also explains the name of this Java 8 implementation of `Future`. A client of this API can invoke it, as shown in the next listing.

Listing 16.5 Using an asynchronous API

```
Shop shop = new Shop("BestShop");
long start = System.nanoTime();
Future<Double> futurePrice = shop.getPriceAsync("my favorite product");
long invocationTime = ((System.nanoTime() - start) / 1_000_000);
System.out.println("Invocation returned after " + invocationTime
                                                + " msecs");
// Do some more tasks, like querying other shops
doSomethingElse();
// while the price of the product is being calculated
try {
    double price = futurePrice.get();
    System.out.printf("Price is %.2f%n", price);
} catch (Exception e) {
    throw new RuntimeException(e);
}
long retrievalTime = ((System.nanoTime() - start) / 1_000_000);
System.out.println("Price returned after " + retrievalTime + " msecs");
```

Query the shop to retrieve the price of a product.

Read the price from the Future or block until it becomes available.

As you can see, the client asks the shop to get the price of a certain product. Because the shop provides an asynchronous API, this invocation almost immediately returns the `Future`, through which the client can retrieve the product's price later. Then the client can perform other tasks, such as querying other shops, instead of remaining blocked, waiting for the first shop to produce the requested result. Later, when the client can do no other meaningful jobs without having the product price, it can invoke `get` on the `Future`. By doing so, the client unwraps the value contained in the `Future` (if the asynchronous task is finished) or remains blocked until that value is available. The output produced by the code in listing 16.5 could be something like this:

```
Invocation returned after 43 msecs
Price is 123.26
Price returned after 1045 msecs
```

You can see that the invocation of the `getPriceAsync` method returns far sooner than when the price calculation eventually finishes. In section 16.4, you learn that it's also possible for the client to avoid any risk of being blocked. Instead, the client can be notified when the `Future` is complete and can execute a callback code, defined through a lambda expression or a method reference, only when the result of the computation is available. For now, we'll address another problem: how to manage an error during the execution of the asynchronous task.

16.2.2 Dealing with errors

The code you've developed so far works correctly if everything goes smoothly. But what happens if the price calculation generates an error? Unfortunately, in this case you get a particularly negative outcome: the exception raised to signal the error remains confined in the thread, which is trying to calculate the product price, and ultimately kills the thread. As a consequence, the client remains blocked forever, waiting for the result of the `get` method to arrive.

The client can prevent this problem by using an overloaded version of the `get` method that also accepts a timeout. It's good practice to use a timeout to prevent similar situations elsewhere in your code. This way, the client at least avoids waiting indefinitely, but when the timeout expires, it's notified with a `TimeoutException`. As a consequence, the client won't have a chance to discover what caused that failure inside the thread that was trying to calculate the product price. To make the client aware of the reason why the shop wasn't able to provide the price of the requested product, you have to propagate the `Exception` that caused the problem inside the `CompletableFuture` through its `completeExceptionally` method. Applying this idea to listing 16.4 produces the code shown in the following listing.

Listing 16.6 Propagating an error inside the `CompletableFuture`

```
public Future<Double> getPriceAsync(String product) {
    CompletableFuture<Double> futurePrice = new CompletableFuture<>();
    new Thread( () -> {
                  try {
                      double price = calculatePrice(product);
                      futurePrice.complete(price);
                  } catch (Exception ex) {
                      futurePrice.completeExceptionally(ex);
                  }
    }).start();
    return futurePrice;
}
```

If the price calculation completed normally, complete the Future with the price.

Otherwise, complete the Future exceptionally with the Exception that caused the failure.

Now the client will be notified with an `ExecutionException` (which takes an `Exception` parameter containing the cause—the original `Exception` thrown by the price calculation method). If that method throws a `RuntimeException` saying that product isn't available, for example, the client gets an `ExecutionException` like the following:

```
Exception in thread "main" java.lang.RuntimeException:
     java.util.concurrent.ExecutionException: java.lang.RuntimeException:
         product not available
    at java89inaction.chap16.AsyncShopClient.main(AsyncShopClient.java:16)
Caused by: java.util.concurrent.ExecutionException: java.lang.RuntimeException:
     product not available
    at java.base/java.util.concurrent.CompletableFuture.reportGet
      (CompletableFuture.java:395)
    at java.base/java.util.concurrent.CompletableFuture.get
      (CompletableFuture.java:1999)
```

```
    at java89inaction.chap16.AsyncShopClient.main(AsyncShopClient.java:14)
Caused by: java.lang.RuntimeException: product not available
    at java89inaction.chap16.AsyncShop.calculatePrice(AsyncShop.java:38)
    at java89inaction.chap16.AsyncShop.lambda$0(AsyncShop.java:33)
    at java.base/java.util.concurrent.CompletableFuture$AsyncSupply.run
     (CompletableFuture.java:1700)
    at java.base/java.util.concurrent.CompletableFuture$AsyncSupply.exec
     (CompletableFuture.java:1692)
    at java.base/java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:283)
    at java.base/java.util.concurrent.ForkJoinPool.runWorker
     (ForkJoinPool.java:1603)
    at java.base/java.util.concurrent.ForkJoinWorkerThread.run
     (ForkJoinWorkerThread.java:175)
```

CREATING A COMPLETABLEFUTURE WITH THE SUPPLYASYNC FACTORY METHOD

Up to now, you've created `CompletableFutures` and completed them programmatically when it seemed convenient to do so, but the `CompletableFuture` class comes with lots of handy factory methods that can make this process far easier and less verbose. The `supplyAsync` method, for example, lets you rewrite the `getPriceAsync` method in listing 16.4 with a single statement, as shown in the next listing.

Listing 16.7 Creating a `CompletableFuture` with the `supplyAsync` factory method

```
public Future<Double> getPriceAsync(String product) {
    return CompletableFuture.supplyAsync(() -> calculatePrice(product));
}
```

The `supplyAsync` method accepts a `Supplier` as argument and returns a `CompletableFuture` that will be asynchronously completed with the value obtained by invoking that `Supplier`. This `Supplier` is run by one of the `Executors` in the `ForkJoinPool`, but you can specify a different `Executor` by passing it as a second argument to the overloaded version of this method. More generally, you can pass an `Executor` to all other `CompletableFuture` factory methods. You use this capability in section 16.3.4, where we demonstrate that using an `Executor` that fits the characteristics of your application can have a positive effect on its performance.

Also note that the `CompletableFuture` returned by the `getPriceAsync` method in listing 16.7 is equivalent to the one you created and completed manually in listing 16.6, meaning that it provides the same error management you carefully added.

For the rest of this chapter, we'll suppose that you have no control of the API implemented by the `Shop` class and that it provides only synchronous blocking methods. This situation typically happens when you want to consume an HTTP API provided by some service. You see how it's still possible to query multiple shops asynchronously, thus avoiding becoming blocked on a single request and thereby increasing the performance and the throughput of your best-price-finder application.

16.3 Making your code nonblocking

You've been asked to develop a best-price-finder application, and all the shops you have to query provide only the same synchronous API implemented as shown at the beginning of section 16.2. In other words, you have a list of shops, like this one:

```
List<Shop> shops = List.of(new Shop("BestPrice"),
                           new Shop("LetsSaveBig"),
                           new Shop("MyFavoriteShop"),
                           new Shop("BuyItAll"));
```

You have to implement a method with the following signature, which, given the name of a product, returns a list of strings. Each string contains the name of a shop and the price of the requested product in that shop, as follows:

```
public List<String> findPrices(String product);
```

Your first idea probably will be to use the `Stream` features you learned in chapters 4, 5, and 6. You may be tempted to write something like this next listing. (Yes, it's good if you're already thinking that this first solution is bad!)

Listing 16.8 A `findPrices` implementation sequentially querying all the shops

```
public List<String> findPrices(String product) {
    return shops.stream()
       .map(shop -> String.format("%s price is %.2f",
                                  shop.getName(), shop.getPrice(product)))
       .collect(toList());
}
```

This solution is straightforward. Now try to put the method `findPrices` to work with the only product you want madly these days: the myPhone27S. In addition, record how long the method takes to run, as shown in the following listing. This information lets you compare the method's performance with that of the improved method you develop later.

Listing 16.9 Checking `findPrices` correctness and performance

```
long start = System.nanoTime();
System.out.println(findPrices("myPhone27S"));
long duration = (System.nanoTime() - start) / 1_000_000;
System.out.println("Done in " + duration + " msecs");
```

The code in listing 16.9 produces output like this:

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47, MyFavoriteShop price
     is 214.13, BuyItAll price is 184.74]
Done in 4032 msecs
```

As you may have expected, the time that the `findPrices` method takes to run is a few milliseconds longer than 4 seconds, because the four shops are queried sequentially

and blocking one after the other, and each shop takes 1 second to calculate the price of the requested product. How can you improve on this result?

16.3.1 Parallelizing requests using a parallel Stream

After reading chapter 7, the first and quickest improvement that should occur to you would be to avoid this sequential computation by using a parallel `Stream` instead of a sequential, as shown in the next listing.

Listing 16.10 Parallelizing the `findPrices` method

```
public List<String> findPrices(String product) {
    return shops.parallelStream()
       .map(shop -> String.format("%s price is %.2f",
                                  shop.getName(), shop.getPrice(product)))
       .collect(toList());
}
```

Use a parallel Stream to retrieve the prices from the different shops in parallel.

Find out whether this new version of `findPrices` is any better by again running the code in listing 16.9:

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47, MyFavoriteShop price
     is 214.13, BuyItAll price is 184.74]
Done in 1180 msecs
```

Well done! It looks like this idea is simple but effective. Now the four shops are queried in parallel, so the code takes a bit more than a second to complete.

Can you do even better? Try to turn all the synchronous invocations to the shops in the `findPrices` method into asynchronous invocations, using what you've learned so far about `CompletableFutures`.

16.3.2 Making asynchronous requests with CompletableFutures

You saw earlier that you can use the factory method `supplyAsync` to create `CompletableFuture` objects. Now use it:

```
List<CompletableFuture<String>> priceFutures =
        shops.stream()
        .map(shop -> CompletableFuture.supplyAsync(
             () -> String.format("%s price is %.2f",
             shop.getName(), shop.getPrice(product))))
        .collect(toList());
```

With this approach, you obtain a `List<CompletableFuture<String>>`, where each `CompletableFuture` in the `List` contains the `String` name of a shop when its computation is complete. But because the `findPrices` method you're trying to reimplement with `CompletableFutures` has to return a `List<String>`, you'll have to wait for the completion of all these futures and extract the values they contain before returning the `List`.

To achieve this result, you can apply a second `map` operation to the original `List<CompletableFuture<String>>`, invoking a `join` on all the futures in the `List` and then waiting for their completion one by one. Note that the `join` method of the `CompletableFuture` class has the same meaning as the `get` method also declared in the `Future` interface, the only difference being that `join` doesn't throw any checked exception. By using `join`, you don't have to bloat the lambda expression passed to this second `map` with a `try/catch` block. Putting everything together, you can rewrite the `findPrices` method as shown in the listing that follows.

Listing 16.11 Implementing the `findPrices` method with `CompletableFutures`

```
public List<String> findPrices(String product) {
    List<CompletableFuture<String>> priceFutures =
            shops.stream()
            .map(shop -> CompletableFuture.supplyAsync(
                         () -> shop.getName() + " price is " +
                               shop.getPrice(product)))
            .collect(Collectors.toList());
    return priceFutures.stream()
            .map(CompletableFuture::join)
            .collect(toList());
}
```

Calculate each price asynchronously with a CompletableFuture.

Wait for the completion of all asynchronous operations.

Note that you use two separate stream pipelines instead of putting the two `map` operations one after the other in the same stream-processing pipeline—and for a good reason. Given the lazy nature of intermediate stream operations, if you'd processed the stream in a single pipeline, you'd have succeeded only in executing all the requests to different shops synchronously and sequentially. The creation of each `CompletableFuture` to interrogate a given shop would start only when the computation of the previous one completed, letting the `join` method return the result of that computation. Figure 16.2 clarifies this important detail.

The top half of figure 16.2 shows that processing the stream with a single pipeline implies that the evaluation order (identified by the dotted line) is sequential. In fact, a new `CompletableFuture` is created only after the former one is completely evaluated. Conversely, the bottom half of the figure demonstrates how first gathering the `CompletableFutures` in a list (represented by the oval) allows all of them to start before waiting for their completion.

Running the code in listing 16.11 to check the performance of this third version of the `findPrices` method, you could obtain output along these lines:

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47, MyFavoriteShop price
     is 214.13, BuyItAll price is 184.74]
Done in 2005 msecs
```

This result is quite disappointing, isn't it? With a runtime of more than 2 seconds, this implementation with `CompletableFutures` is faster than the original naïve sequential

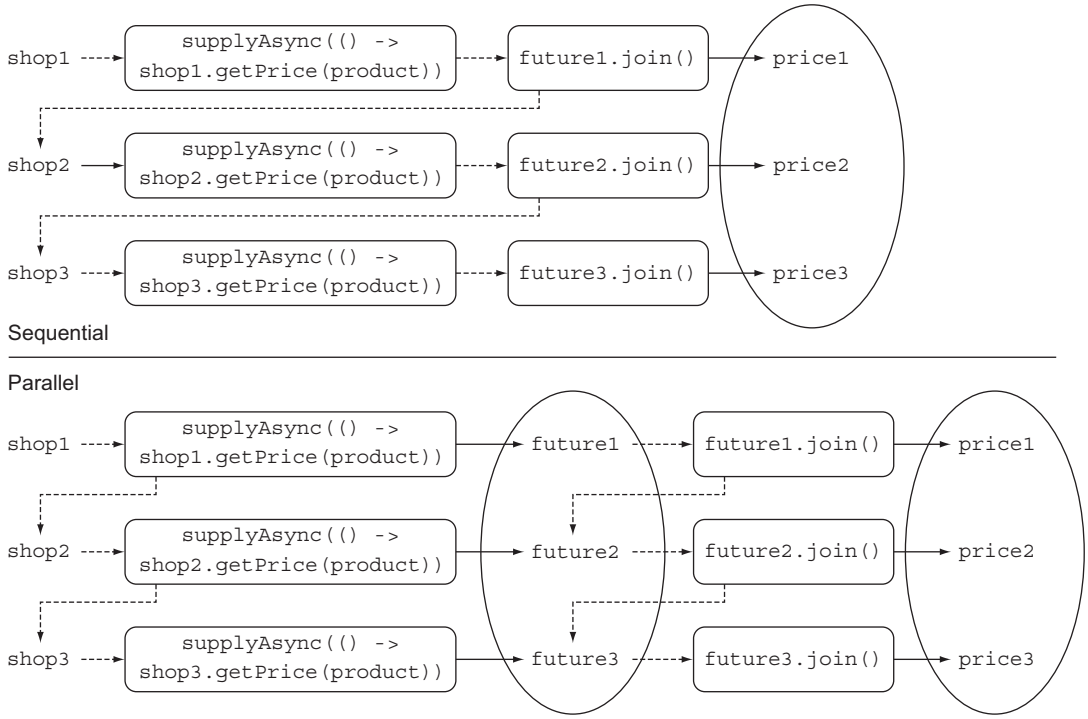

Figure 16.2 Why `Stream`'s laziness causes a sequential computation and how to avoid it

and blocking implementation from listing 16.8. But it's also almost twice as slow as the previous implementation using a parallel stream. It's even more disappointing considering the fact that you obtained the parallel stream version with a trivial change to the sequential version.

The newer version with `CompletableFutures` required quite a bit of work. But is using `CompletableFutures` in this scenario a waste of time? Or are you overlooking something important? Take a few minutes before going forward, particularly recalling that you're testing the code samples on a machine that's capable of running four threads in parallel.[1]

16.3.3 Looking for the solution that scales better

The parallel stream version performs so well only because it can run four tasks in parallel, so it's able to allocate exactly one thread for each shop. What happens if you decide to add a fifth shop to the list of shops crawled by your best-price-finder application? Not

[1] If you're using a machine that's capable of running more threads in parallel (say, eight), you need more shops and processes in parallel to reproduce the behavior shown in these pages.

surprisingly, the sequential version requires a bit more than 5 seconds to run, as shown in the following output:

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47, MyFavoriteShop price
     is 214.13, BuyItAll price is 184.74, ShopEasy price is 166.08]
Done in 5025 msecs
```

The output of the program using a sequential stream

Unfortunately, the parallel stream version also requires a whole second more than before, because all four threads that it can run in parallel (available in the common thread pool) are now busy with the first four shops. The fifth query has to wait for the completion of one of the former operations to free a thread, as follows:

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47, MyFavoriteShop price
     is 214.13, BuyItAll price is 184.74, ShopEasy price is 166.08]
Done in 2167 msecs
```

The output of the program using a parallel stream

What about the `CompletableFuture` version? Give it a try with the fifth shop:

```
[BestPrice price is 123.26, LetsSaveBig price is 169.47, MyFavoriteShop price
     is 214.13, BuyItAll price is 184.74, ShopEasy price is 166.08]
Done in 2006 msecs
```

The output of the program using CompletableFutures

The `CompletableFuture` version seems to be a bit faster than the one using parallel stream, but this version isn't satisfying either. If you try to run your code with nine shops, the parallel stream version takes 3143 milliseconds, whereas the `CompletableFuture` cersion requires 3009 milliseconds. The versions look equivalent for a good reason: both internally use the same common pool that by default has a fixed number of threads equal to the one returned by `Runtime.getRuntime().availableProcessors()`. Nevertheless, the `CompletableFutures` version has an advantage: by contrast with the parallel `Streams` API, it allows you to specify a different `Executor` to submit tasks to. You can configure this `Executor`, and size its thread pool, in a way that better fits the requirements of your application. In the next section, you translate this better level of configurability into practical performance gain for your application.

16.3.4 Using a custom Executor

In this case, a sensible choice seems to be to create an `Executor` with a number of threads in its pool that takes into account the actual workload you could expect in your application. How do you size this `Executor` correctly?

Sizing thread pools

In the great book *Java Concurrency in Practice* (Addison-Wesley, 2006; http://jcip.net), Brian Goetz and his co-authors give some advice on finding the optimal size for a thread pool. This advice is important because if the number of threads in the pool is too big, the threads end up competing for scarce CPU and memory resources, wasting their time performing context switching. Conversely, if this number is too small (as it likely is in your application), some of the cores of the CPU will remain underused. Goetz suggests that you can calculate the right pool size to approximate a desired CPU use rate with the following formula:

$$N^{threads} = N^{CPU} * U^{CPU} * (1 + W/C)$$

In this formula, NCPU is the number of cores, available through `Runtime.getRuntime().availableProcessors()`

- U^{CPU} is the target CPU use (between 0 and 1).
- W/C is the ratio of wait time to compute time.

The application is spending about 99 percent of its time waiting for the shops' responses, so you could estimate a W/C ratio of 100. If your target is 100 percent CPU use, you should have a pool with 400 threads. In practice, it's wasteful to have more threads than shops, because you'll have threads in your pool that are never used. For this reason, you need to set up an `Executor` with a fixed number of threads equal to the number of shops you have to query, so that you have one thread for each shop. Also set an upper limit of 100 threads to avoid a server crash for a larger number of shops, as shown in the following listing.

Listing 16.12 A custom `Executor` fitting the best-price-finder application

```
private final Executor executor =
        Executors.newFixedThreadPool(Math.min(shops.size(), 100),
                                     (Runnable r) -> {
            Thread t = new Thread(r);
            t.setDaemon(true);
            return t;
        }
);
```

Use daemon threads, which don't prevent the termination of the program.

Create a thread pool with a number of threads equal to the minimum of 100 and the number of shops.

Note that you're creating a pool made of daemon threads. A Java program can't terminate or exit while a normal thread is executing, so a leftover thread waiting for a never-satisfiable event causes problems. By contrast, marking a thread as a daemon means that it can be killed on program termination. There's no performance difference. Now you can pass the new `Executor` as the second argument of the `supplyAsync`

factory method. In addition, now create the `CompletableFuture` that retrieves the price of the requested product from a given shop as follows:

```
CompletableFuture.supplyAsync(() -> shop.getName() + " price is " +
                                    shop.getPrice(product), executor);
```

After this improvement, the `CompletableFutures` solution takes 1021 milliseconds to process five shops and 1022 milliseconds to process nine shops. This trend carries on until the number of shops reaches the threshold of 400 that you calculated earlier. This example demonstrates the fact that it's a good idea to create an `Executor` that fits the characteristics of your application and to use `CompletableFutures` to submit tasks to it. This strategy is almost always effective and something to consider when you make intensive use of asynchronous operations.

Parallelism: via Streams or CompletableFutures?

You've seen two ways to do parallel computing on a collection: convert the collection to a parallel stream and use operations such as `map` on it, or iterate over the collection and spawn operations within a `CompletableFuture`. The latter technique provides more control by resizing thread pools, which ensures that your overall computation doesn't block because all your (fixed number of) threads are waiting for I/O.

Our advice for using these APIs is as follows:

- If you're doing computation-heavy operations with no I/O, the `Stream` interface provides the simplest implementation and the one that's likely to be most efficient. (If all threads are compute-bound, there's no point in having more threads than processor cores.)
- If your parallel units of work involve waiting for I/O (including network connections), the `CompletableFutures` solution provides more flexibility and allows you to match the number of threads to the wait/computer (W/C) ratio, as discussed previously. Another reason to avoid using parallel streams when I/O waits are involved in the stream-processing pipeline is that the laziness of streams can make it harder to reason about when the waits happen.

You've learned how to take advantage of `CompletableFutures` to provide an asynchronous API to your clients and to function as the client of a synchronous but slow server, but you performed only a single time-consuming operation in each `Future`. In the next section, you use `CompletableFutures` to pipeline multiple asynchronous operations in a declarative style similar to what you learned by using the Streams API.

16.4 Pipelining asynchronous tasks

Suppose that all the shops have agreed to use a centralized discount service. This service uses five discount codes, each of which has a different discount percentage. You represent this idea by defining a `Discount.Code` enumeration, as shown in the next listing.

Listing 16.13 An enumeration defining the discount codes

```
public class Discount {
    public enum Code {
        NONE(0), SILVER(5), GOLD(10), PLATINUM(15), DIAMOND(20);
        private final int percentage;
        Code(int percentage) {
            this.percentage = percentage;
        }
    }
    // Discount class implementation omitted, see Listing 16.14
}
```

Also suppose that the shops have agreed to change the format of the result of the `getPrice` method, which now returns a `String` in the format `ShopName:price:DiscountCode`. Your sample implementation returns a random `Discount.Code` together with the random price already calculated, as follows:

```
public String getPrice(String product) {
    double price = calculatePrice(product);
    Discount.Code code = Discount.Code.values()[
                            random.nextInt(Discount.Code.values().length)];
    return String.format("%s:%.2f:%s", name, price, code);
}
private double calculatePrice(String product) {
    delay();
    return random.nextDouble() * product.charAt(0) + product.charAt(1);
}
```

Invoking `getPrice` might then return a `String` such as

```
BestPrice:123.26:GOLD
```

16.4.1 Implementing a discount service

Your best-price-finder application should now obtain the prices from the shops; parse the resulting `String`s; and, for each `String`, query the discount server's needs. This process determines the final discounted price of the requested product. (The actual discount percentage associated with each discount code could change, which is why you query the server each time.) The parsing of the `String`s produced by the shop is encapsulated in the following `Quote` class:

```
public class Quote {
    private final String shopName;
    private final double price;
    private final Discount.Code discountCode;
    public Quote(String shopName, double price, Discount.Code code) {
        this.shopName = shopName;
        this.price = price;
        this.discountCode = code;
    }
```

```
    public static Quote parse(String s) {
        String[] split = s.split(":");
        String shopName = split[0];
        double price = Double.parseDouble(split[1]);
        Discount.Code discountCode = Discount.Code.valueOf(split[2]);
        return new Quote(shopName, price, discountCode);
    }
    public String getShopName() { return shopName; }
    public double getPrice() { return price; }
    public Discount.Code getDiscountCode() { return discountCode; }
}
```

You can obtain an instance of the `Quote` class—which contains the name of the shop, the nondiscounted price, and the discount code—by passing the `String` produced by a shop to the static `parse` factory method.

The `Discount` service also has an `applyDiscount` method that accepts a `Quote` object and returns a `String` stating the discounted price for the shop that produced that quote, as shown in the following listing.

Listing 16.14 Listing 16.14 The `Discount` service

```
public class Discount {
    public enum Code {
        // source omitted ...
    }
    public static String applyDiscount(Quote quote) {
        return quote.getShopName() + " price is " +
               Discount.apply(quote.getPrice(),
                              quote.getDiscountCode());
    }
    private static double apply(double price, Code code) {
        delay();
        return format(price * (100 - code.percentage) / 100);
    }
}
```

Apply the discount code to the original price.

Simulate a delay in the Discount service response.

16.4.2 Using the Discount service

Because the `Discount` service is a remote service, you again add a simulated delay of 1 second to it, as shown in the next listing. As you did in section 16.3, first try to reimplement the `findPrices` method to fit these new requirements in the most obvious (but, sadly, sequential and synchronous) way.

Listing 16.15 Simplest `findPrices` implementation that uses the `Discount` service

```
public List<String> findPrices(String product) {
    return shops.stream()
            .map(shop -> shop.getPrice(product))
            .map(Quote::parse)
            .map(Discount::applyDiscount)
            .collect(toList());
}
```

Transform the Strings returned by the shops in Quote objects.

Retrieve the nondiscounted price from each shop.

Contact the Discount service to apply the discount on each Quote.

You obtain the desired result by pipelining three `map` operations on the stream of shops:

- The first operation transforms each shop into a `String` that encodes the price and discount code of the requested product for that shop.
- The second operation parses those `String`s, converting each of them in a `Quote` object.
- The third operation contacts the remote `Discount` service, which calculates the final discounted price and returns another `String` containing the name of the shop with that price.

As you might imagine, the performance of this implementation is far from optimal. But try to measure it as usual by running your benchmark:

```
[BestPrice price is 110.93, LetsSaveBig price is 135.58, MyFavoriteShop price
     is 192.72, BuyItAll price is 184.74, ShopEasy price is 167.28]
Done in 10028 msecs
```

As expected, this code takes 10 seconds to run, because the 5 seconds required to query the five shops sequentially is added to the 5 seconds consumed by the discount service in applying the discount code to the prices returned by the five shops. You already know that you can improve this result by converting the stream to a parallel one. But you also know (from section 16.3) that this solution doesn't scale well when you increase the number of shops to be queried, due to the fixed common thread pool on which streams rely. Conversely, you learned that you can better use your CPU by defining a custom `Executor` that schedules the tasks performed by the `CompletableFuture`s.

16.4.3 Composing synchronous and asynchronous operations

In this section, you try to reimplement the `findPrices` method asynchronously, again using the features provided by `CompletableFuture`. This next listing shows the code. Don't worry if something looks unfamiliar; we explain the code in this section.

Listing 16.16 Implementing the `findPrices` method with `CompletableFuture`s

Transform the String returned by a shop into a Quote object when it becomes available.

Asynchronously retrieve the nondiscounted price from each shop.

```
public List<String> findPrices(String product) {
    List<CompletableFuture<String>> priceFutures =
        shops.stream()
             .map(shop -> CompletableFuture.supplyAsync(
                                 () -> shop.getPrice(product), executor))
             .map(future -> future.thenApply(Quote::parse))
             .map(future -> future.thenCompose(quote ->
                         CompletableFuture.supplyAsync(
                           () -> Discount.applyDiscount(quote), executor)))
                .collect(toList());
    return priceFutures.stream()
```

Compose the resulting Future with another asynchronous task, applying the discount code.

```
                .map(CompletableFuture::join)
                .collect(toList());
}
```

Wait for all the Futures in the stream to be completed and extract their respective results.

Things look a bit more complex this time, so try to understand what's going on step-by-step. Figure 16.3 depicts the sequence of these three transformations.

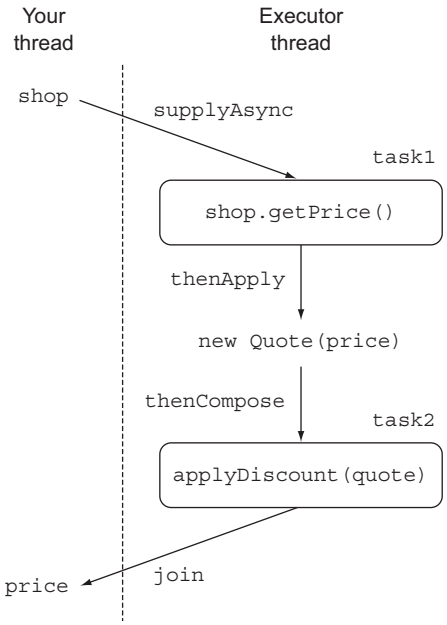

Figure 16.3 Composing synchronous operations and asynchronous tasks

You're performing the same three `map` operations that you did in the synchronous solution of listing 16.15, but you make those operations asynchronous when necessary, using the feature provided by the `CompletableFuture` class.

GETTING THE PRICES

You've seen the first of these three operations in various examples in this chapter; you query the shop asynchronously by passing a lambda expression to the `supplyAsync` factory method. The result of this first transformation is a `Stream<CompletableFuture<String>>`, where each `CompletableFuture` contains, when complete, the `String` returned by the corresponding shop. Note that you configure the `CompletableFutures` with the custom `Executor` developed in listing 16.12.

PARSING THE QUOTES

Now you have to convert those `Strings` to `Quotes` with a second transformation. But because this parsing operation isn't invoking any remote service or doing any I/O in general, it can be performed almost instantaneously and can be done synchronously without introducing any delay. For this reason, you implement this second transformation by

invoking the `thenApply` method on the `CompletableFutures` produced by the first step and passing to it a `Function` converting a `String` to an instance of `Quote`.

Note that using the `thenApply` method doesn't block your code until the `CompletableFuture` on which you're invoking it is complete. When the `CompletableFuture` finally completes, you want to transform the value that it contains by using the lambda expression passed to the `thenApply` method, thus transforming each `CompletableFuture<String>` in the stream into a corresponding `CompletableFuture<Quote>`. You can see this process as building a recipe that specifies what to do with the result of the `CompletableFuture`, as when you worked with a stream pipeline.

Composing the futures for calculating the discounted price

The third `map` operation involves contacting the remote `Discount` service to apply the appropriate discount percentage to the nondiscounted prices received from the shops. This transformation is different from the previous one because it has to be executed remotely (or, in this case, has to simulate the remote invocation with a delay), and for this reason, you also want to perform it asynchronously.

To achieve this goal, as you did with the first invocation of `supplyAsync` with `getPrice`, you pass this operation as a lambda expression to the `supplyAsync` factory method, which returns another `CompletableFuture`. At this point you have two asynchronous operations, modeled with two distinct `CompletableFutures`, that you want to perform in a cascade:

- Retrieve the price from a shop and then transform it into a `Quote`.
- Take this `Quote` and pass it to the `Discount` service to obtain the final discounted price.

The Java 8 `CompletableFuture` API provides the `thenCompose` method specifically for this purpose, allowing you to pipeline two asynchronous operations, passing the result of the first operation to the second operation when it becomes available. In other words, you can compose two `CompletableFutures` by invoking the `thenCompose` method on the first `CompletableFuture` and passing to it a `Function`. This `Function` has as an argument the value returned by that first `CompletableFuture` when it completes, and it returns a second `CompletableFuture` that uses the result of the first as input for its computation. Note that with this approach, while the `Futures` are retrieving the quotes from the shops, the main thread can perform other useful operations, such as responding to UI events.

Collecting the elements of the `Stream` resulting from these three `map` operations into a `List`, you obtain a `List<CompletableFuture<String>>`. Finally, you can wait for the completion of those `CompletableFutures` and extract their values by using `join`, exactly as you did in listing 16.11. This new version of the `findPrices` method implemented in listing 16.8 might produce output like this:

```
[BestPrice price is 110.93, LetsSaveBig price is 135.58, MyFavoriteShop price
     is 192.72, BuyItAll price is 184.74, ShopEasy price is 167.28]
Done in 2035 msecs
```

The thenCompose method you used in listing 16.16, like other methods of the CompletableFuture class, has a variant with an Async suffix, thenComposeAsync. In general, a method without the Async suffix in its name executes its task in the same thread as the previous task, whereas a method terminating with Async always submits the succeeding task to the thread pool, so each of the tasks can be handled by a different thread. In this case, the result of the second CompletableFuture depends on the first, so it makes no difference to the final result or to its broad-brush timing whether you compose the two CompletableFutures with one or the other variant of this method. You chose to use the one with thenCompose only because it's slightly more efficient due to less thread-switching overhead. Note, however, that it may not always be clear which thread is being used, especially if you run an application that manages its own thread pool (such as Spring).

16.4.4 Combining two CompletableFutures: dependent and independent

In listing 16.16, you invoked the thenCompose method on one CompletableFuture and passed to it a second CompletableFuture, which needed as input the value resulting from the execution of the first. In another common case, you need to combine the results of the operations performed by two independent CompletableFutures, and you don't want to wait for the first to complete before starting the second.

In situations like this one, use the thenCombine method. This method takes as a second argument a BiFunction, which defines how the results of the two CompletableFutures are to be combined when both become available. Like thenCompose, the thenCombine method comes with an Async variant. In this case, using the thenCombineAsync method causes the combination operation defined by the BiFunction to be submitted to the thread pool and then executed asynchronously in a separate task.

Turning to this chapter's running example, you may know that one of the shops provides prices in € (EUR), but you always want to communicate them to your customers in $(USD). You can asynchronously ask the shop the price of a given product *and separately* retrieve, from a remote exchange-rate service, the current exchange rate between € and $. After both requests have completed, you can combine the results by multiplying the price by the exchange rate. With this approach, you obtain a third CompletableFuture that completes when the results of the two CompletableFutures are both available and have been combined by means of the BiFunction, as shown in the following listing.

Listing 16.17 Combining two independent CompletableFutures

```
Future<Double> futurePriceInUSD =
        CompletableFuture.supplyAsync(() -> shop.getPrice(product))
        .thenCombine(
            CompletableFuture.supplyAsync(
                () ->  exchangeService.getRate(Money.EUR, Money.USD)),
            (price, rate) -> price * rate
        ));
```

Create a first task querying the shop to obtain the price of a product.

Combine the price and exchange rate by multiplying them.

Create a second independent task to retrieve the conversion rate between USD and EUR.

Here, because the combination operation is a simple multiplication, performing it in a separate task would have been a waste of resources, so you need to use the `thenCombine` method instead of its asynchronous `thenCombineAsync` counterpart. Figure 16.4 shows how the tasks created in listing 16.17 are executed on the different threads of the pool and how their results are combined.

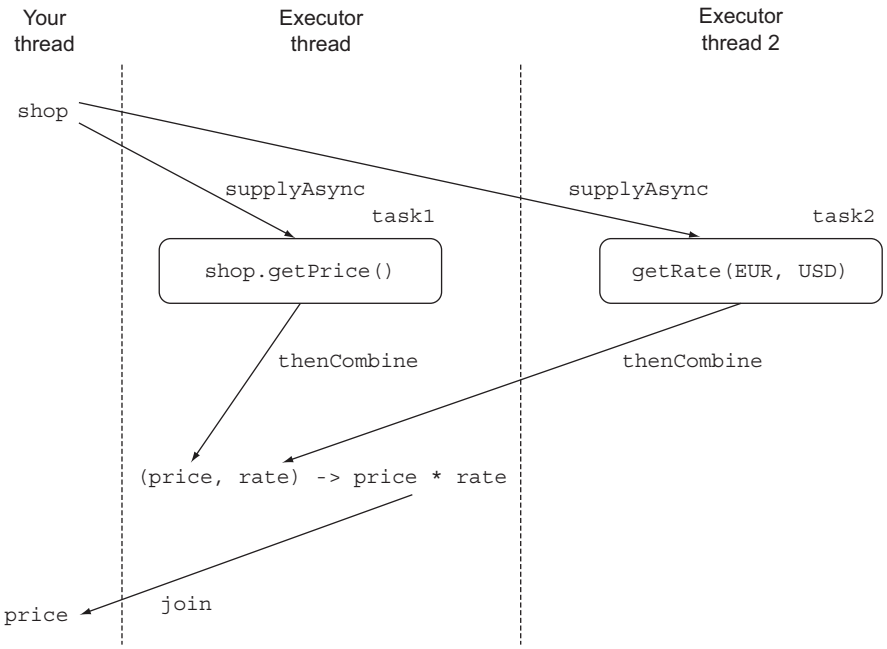

Figure 16.4 Combining two independent asynchronous tasks

16.4.5 Reflecting on Future vs. CompletableFuture

The last two examples in listings 16.16 and 16.17 clearly show one of the biggest advantages of `CompletableFutures` over the other pre-Java 8 `Future` implementations. `CompletableFutures` use lambda expressions to provide a declarative API. This API allows you to easily combine and compose various synchronous and asynchronous tasks to perform a complex operation in the most effective way. To get a more tangible idea of the code-readability benefits of `CompletableFuture`, try to obtain the result of listing 16.17 purely in Java 7. The next listing shows you how.

Listing 16.18 Combining two `Futures` in Java 7

Create an ExecutorService allowing you to submit tasks to a thread pool.

```
ExecutorService executor = Executors.newCachedThreadPool();
final Future<Double> futureRate = executor.submit(new Callable<Double>() {
        public Double call() {
```

```
            return exchangeService.getRate(Money.EUR, Money.USD);
        }});
Future<Double> futurePriceInUSD = executor.submit(new Callable<Double>() {
        public Double call() {
            double priceInEUR = shop.getPrice(product);
            return priceInEUR * futureRate.get();
        }});
```

Find the price of the requested product for a given shop in a second Future.

Multiply the price and exchange rate in the same Future used to find the price.

Create a Future retrieving the exchange rate between EUR and USD.

In listing 16.18, you create a first `Future`, submitting a `Callable` to an `Executor` querying an external service to find the exchange rate between EUR and USD. Then you create a second `Future`, retrieving the price in EUR of the requested product for a given shop. Finally, as you did in listing 16.17, you multiply the exchange rate by the price in the same future that also queried the shop to retrieve the price in EUR. Note that using `thenCombineAsync` instead of `thenCombine` in listing 16.17 would have been equivalent to performing the price by rate multiplication in a third `Future` in listing 16.18. The difference between these two implementations may seem to be small only because you're combining two `Futures`.

16.4.6 Using timeouts effectively

As mentioned in section 16.2.2, it's always a good idea to specify a timeout when trying to read the value calculated by a `Future` to avoid being blocked indefinitely while waiting for the computation of that value. Java 9 introduced a couple of convenient methods that enrich the timeout capabilities provided by the `CompletableFutures`. The `orTimeout` method uses a `ScheduledThreadExecutor` to complete the `CompletableFuture` with a `TimeoutException` after the specified timeout has elapsed, and it returns another `CompletableFuture`. By using this method, you can further chain your computation pipeline and deal with the `TimeoutException` by providing a friendly message back. You can add a timeout to the `Future` in listing 16.17 and make it throw a `TimeoutException` if not completed after 3 seconds by adding this method at the end of the methods chain, as shown in the next listing. The timeout duration should match your business requirements, of course.

Listing 16.19 Adding a timeout to `CompletableFuture`

```
Future<Double> futurePriceInUSD =
        CompletableFuture.supplyAsync(() -> shop.getPrice(product))
        .thenCombine(
            CompletableFuture.supplyAsync(
                () ->  exchangeService.getRate(Money.EUR, Money.USD)),
            (price, rate) -> price * rate
        ))
        .orTimeout(3, TimeUnit.SECONDS);
```

Make the Future throw a TimeoutException if not completed after 3 seconds. Asynchronous timeout management was added in Java 9.

Sometimes, it's also acceptable to use a default value in case a service is momentarily unable to respond in a timely manner. You might decide that in listing 16.19, you want to wait for the exchange service to provide the current EUR-to-USD exchange rate for no more than 1 second, but if the request takes longer to complete, you don't want to abort the whole the computation with an `Exception`. Instead, you can fall back by using a predefined exchange rate. You can easily add this second kind of timeout by using the `completeOnTimeout` method, also introduced in Java 9 (the following listing).

Listing 16.20 Completing a `CompletableFuture` with a default value after a timeout

```
Future<Double> futurePriceInUSD =
        CompletableFuture.supplyAsync(() -> shop.getPrice(product))
        .thenCombine(
            CompletableFuture.supplyAsync(
                () ->  exchangeService.getRate(Money.EUR, Money.USD))
              .completeOnTimeout(DEFAULT_RATE, 1, TimeUnit.SECONDS),
            (price, rate) -> price * rate
        ))
        .orTimeout(3, TimeUnit.SECONDS);
```

Use a default exchange rate if the exchange service doesn't provide a result in 1 second.

Like the `orTimeout` method, the `completeOnTimeOut` method returns a `CompletableFuture`, so you can chain it with other `CompletableFuture` methods. To recap, you've configured two kinds of timeouts: one that makes the whole computation fail if it takes more than 3 seconds, and one that expires in 1 second but completes the `Future` with a predetermined value instead of causing a failure.

You're almost finished with your best-price-finder application, but one ingredient is still missing. You'd like to show your users the prices provided by the shops as soon as they become available (as car insurance and flight-comparison websites typically do), instead of waiting for all the price requests to complete, as you've done up to now. In the next section, you discover how to achieve this goal by reacting to the completion of a `CompletableFuture` instead of invoking `get` or `join` on it and thereby remaining blocked until the `CompletableFuture` itself completes.

16.5 Reacting to a CompletableFuture completion

In all the code examples you've seen in this chapter, you've simulated methods that do remote invocations with a 1-second delay in their response. In a real-world scenario, the remote services you need to contact from your application are likely to have unpredictable delays caused by everything from server load to network delays, and perhaps by how valuable the server regards your application's business to be compared with that of applications that pay more per query.

For these reasons, it's likely the prices of the products you want to buy will be available for some shops far earlier than for others. In the next listing, you simulate

this scenario by introducing a random delay of 0.5 to 2.5 seconds, using the `randomDelay` method instead of the `delay` method that waits 1 second.

Listing 16.21 A method to simulate a random delay between 0.5 and 2.5 seconds

```
private static final Random random = new Random();
public static void randomDelay() {
    int delay = 500 + random.nextInt(2000);
    try {
        Thread.sleep(delay);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
```

Up to now, you've implemented the `findPrices` method so that it shows the prices provided by the shops only when all of them are available. Now you want to have the best-price-finder application display the price for a given shop as soon as it becomes available without waiting for the slowest one (which may even time out). How can you achieve this further improvement?

16.5.1 Refactoring the best-price-finder application

The first thing to avoid is waiting for the creation of a `List` that already contains all the prices. You need to work directly with the stream of `CompletableFutures`, in which each `CompletableFuture` is executing the sequence of operations necessary for a given shop. In the next listing, you refactor the first part of the implementation from listing 16.16 into a `findPricesStream` method to produce this stream of `CompletableFutures`.

Listing 16.22 Refactoring the `findPrices` method to return a stream of `Futures`

```
public Stream<CompletableFuture<String>> findPricesStream(String product) {
    return shops.stream()
            .map(shop -> CompletableFuture.supplyAsync(
                                    () -> shop.getPrice(product), executor))
            .map(future -> future.thenApply(Quote::parse))
            .map(future -> future.thenCompose(quote ->
                    CompletableFuture.supplyAsync(
                        () -> Discount.applyDiscount(quote), executor)));
}
```

At this point, you add a fourth `map` operation on the `Stream` returned by the `findPricesStream` method to the three already performed inside that method. This new operation registers an action on each `CompletableFuture`; this action consumes the value of the `CompletableFuture` as soon as it completes. The Java 8 `CompletableFuture` API provides this feature via the `thenAccept` method, which takes as an argument a `Consumer` of the value with which it completes. In this case, this value is the

String returned by the discount services and containing the name of a shop together with the discounted price of the requested product for that shop. The only action that you want to perform to consume this value is to print it:

```
findPricesStream("myPhone").map(f -> f.thenAccept(System.out::println));
```

As you've seen for the thenCompose and thenCombine methods, the thenAccept method has an Async variant named thenAcceptAsync. The Async variant schedules the execution of the Consumer passed to it on a new thread from the thread pool instead of performing it directly, using the same thread that completed the CompletableFuture. Because you want to avoid an unnecessary context switch, and because (more important) you want to react to the completion of the CompletableFuture as soon as possible instead waiting for a new thread to be available, you don't use this variant here.

Because the thenAccept method already specifies how to consume the result produced by the CompletableFuture when it becomes available, it returns a CompletableFuture<Void>. As a result, the map operation returns a Stream<CompletableFuture<Void>>. You can't do much with a CompletableFuture<Void> except wait for its completion, but this is exactly what you need. You also want to give the slowest shop a chance to provide its response and print its returned price. To do so, you can put all the CompletableFuture<Void>s of the stream into an array and then wait for all of them to complete, as shown in this next listing.

Listing 16.23 Reacting to CompletableFuture completion

```
CompletableFuture[] futures = findPricesStream("myPhone")
        .map(f -> f.thenAccept(System.out::println))
        .toArray(size -> new CompletableFuture[size]);
CompletableFuture.allOf(futures).join();
```

The allOf factory method takes as input an array of CompletableFutures and returns a CompletableFuture<Void> that's completed only when all the CompletableFutures passed have completed. Invoking join on the CompletableFuture returned by the allOf method provides an easy way to wait for the completion of all the CompletableFutures in the original stream. This technique is useful for the best-price-finder application because it can display a message such as All shops returned results or timed out so that a user doesn't keep wondering whether more prices might become available.

In other applications, you may want to wait for the completion of only one of the CompletableFutures in an array, perhaps if you're consulting two currency-exchange servers and are happy to take the result of the first to respond. In this case, you can use the anyOf factory method. As a matter of detail, this method takes as input an array of CompletableFutures and returns a CompletableFuture<Object> that completes with the same value as the first-to-complete CompletableFuture.

16.5.2 Putting it all together

As discussed at the beginning of section 16.5, now suppose that all the methods simulating a remote invocation use the `randomDelay` method of listing 16.21, introducing a random delay distributed between 0.5 and 2.5 seconds instead of a delay of 1 second. Running the code in listing 16.23 with this change, you see that the prices provided by the shops don't appear all at the same time, as happened before, but are printed incrementally as soon as the discounted price for a given shop is available. To make the result of this change more obvious, the code is slightly modified to report a timestamp showing the time taken for each price to be calculated:

```
long start = System.nanoTime();
CompletableFuture[] futures = findPricesStream("myPhone27S")
        .map(f -> f.thenAccept(
             s -> System.out.println(s + " (done in " +
                  ((System.nanoTime() - start) / 1_000_000) + " msecs)")))
        .toArray(size -> new CompletableFuture[size]);
CompletableFuture.allOf(futures).join();
System.out.println("All shops have now responded in "
                   + ((System.nanoTime() - start) / 1_000_000) + " msecs");
```

Running this code produces output similar to the following:

```
BuyItAll price is 184.74 (done in 2005 msecs)
MyFavoriteShop price is 192.72 (done in 2157 msecs)
LetsSaveBig price is 135.58 (done in 3301 msecs)
ShopEasy price is 167.28 (done in 3869 msecs)
BestPrice price is 110.93 (done in 4188 msecs)
All shops have now responded in 4188 msecs
```

You can see that, due to the effect of the random delays, the first price now prints more than twice as fast as the last!

16.6 Road map

Chapter 17 explores the Java 9 Flow API, which generalizes the idea of `CompletableFuture` (one-shot, either computing or terminated-with-a-value) by enabling computations to produce a series of values before optionally terminating.

Summary

- Executing relatively long-lasting operations by using asynchronous tasks can increase the performance and responsiveness of your application, especially if it relies on one or more remote external services.
- You should consider providing an asynchronous API to your clients. You can easily implement one by using `CompletableFutures` features.
- A `CompletableFuture` allows you to propagate and manage errors generated within an asynchronous task.
- You can asynchronously consume from a synchronous API by wrapping its invocation in a `CompletableFuture`.

- You can compose or combine multiple asynchronous tasks when they're independent and when the result of one of them is used as the input to another.
- You can register a callback on a `CompletableFuture` to reactively execute some code when the `Future` completes and its result becomes available.
- You can determine when all values in a list of `CompletableFutures` have completed, or you can wait for only the first to complete.
- Java 9 added support for asynchronous timeouts on `CompletableFuture` via the `orTimeout` and `completeOnTimeout` methods.