Chapter 10. Concurrent Programming

Topics in This Chapter

Java was one of the first mainstream programming languages with built-in support for concurrent programming. Early Java programmers were enthusiastic about how easy it was to load images in background threads or implement a web server that serves multiple requests concurrently. At the time, the focus was on keeping a processor busy while some tasks spend their time waiting for the network. Nowadays, most computers have multiple processors or cores, and programmers worry about keeping them all busy.

In this chapter, you will learn how to divide computations into concurrent tasks and how to execute them safely. My focus is on the needs of application programmers, not system programmers who write web servers or middleware.

For that reason, I arranged the information in this chapter so that I can, as much as possible, first show you the tools that you should be using in your work. I cover the low-level constructs later in the chapter. It is useful to know about these low-level details so that you get a feel for the costs of certain operations. But it is best to leave low-level thread programming to the experts. If you want to become one of them, I highly recommend the excellent book *Java Concurrency in Practice* by Brian Goetz et al. [Addison-Wesley, 2006].

The key points of this chapter are:

1. A `Runnable` describes a task that can be executed asynchronously but does

not return a result.

2. An `ExecutorService` schedules tasks instances for execution.
3. A `Callable` describes a task that can be executed asynchronously and yields a result.
4. You can submit one or more `Callable` instances to an `ExecutorService` and combine the results when they are available.
5. When multiple threads operate on shared data without synchronization, the result is unpredictable.
6. Prefer using parallel algorithms and threadsafe data structures over programming with locks.
7. Parallel streams and array operations automatically and safely parallelize computations.
8. A `ConcurrentHashMap` is a threadsafe hash table that allows atomic update of entries.
9. You can use `AtomicLong` for a lock-free shared counter, or use `LongAdder` if contention is high.
10. A lock ensures that only one thread at a time executes a critical section.
11. An interruptible task should terminate when the interrupted flag is set or an `InterruptedException` occurs.
12. A long-running task should not block the user-interface thread of a program, but progress and final updates need to occur in the user-interface thread.
13. The `Process` class lets you execute a command in a separate process and interact with the input, output, and error streams.

10.1 Concurrent Tasks

When you design a concurrent program, you need to think about the tasks that can be run together. In the following sections, you will see how to execute tasks concurrently.

10.1.1 Running Tasks

In Java, the `Runnable` interface describes a task you want to run, perhaps concurrently with others.

```
public interface Runnable {
void run();
}
```

Like all methods, the `run` method is executed in a *thread*. A thread is a mechanism for executing a sequence of instructions, usually provided by the operating system. Multiple threads run concurrently, by using separate processors or different time slices on the same processor.

If you want to execute a `Runnable` in a separate thread, you could spawn a thread just for this `Runnable`, and you will see how to do that in Section 10.8.1. "Starting a Thread" (page 363). But in practice, it doesn't usually make sense to have a one-to-one relationship between tasks and threads. When tasks are short-lived, you want to run many of them on the same thread, so you don't waste the time it takes to start a thread. When your tasks are computationally intensive, you just want one thread per processor instead of one thread per task, to avoid the overhead of switching among threads. You do not want to think of these issues when you design tasks, and therefore, it is best to separate tasks and task scheduling.

In the Java concurrency library, an *executor service* schedules and executes tasks, choosing the threads on which to run them.

Click here to view code image

```
Runnable task = () -> { ... };
ExecutorService executor = ...;
executor.execute(task);
```

The `Executors` class has factory methods for executor services with different scheduling policies. The call

Click here to view code image

```
exec = Executors.newCachedThreadPool();
```

yields an executor service optimized for programs with many tasks that are short lived or spend most of their time waiting. Each task is executed on an idle thread if possible, but a new thread is allocated if all threads are busy. There is no bound on the number of concurrent threads. Threads that are idle for an extended time are terminated.

The call

Click here to view code image

```
exec = Executors.newFixedThreadPool(nthreads);
```

yield a pool with a fixed number of threads. When you submit a task, it is queued up until a thread becomes available. This is a good choice to use for computationally intensive tasks, or to limit the resource consumption of a service. You can derive the number of threads from the number of available processors, which you obtain as

Click here to view code image

```
int processors = Runtime.getRuntime().availableProcessors();
```

Now go ahead and run the concurrency demo program in the book's companion code. It runs two tasks concurrently.

Click here to view code image

```
public static void main(String[] args) {
Runnable hellos = () -> {
for (int i = 1; i <= 1000; i++)
System.out.println("Hello " + i);
};
Runnable goodbyes = () -> {
for (int i = 1; i <= 1000; i++)
System.out.println("Goodbye " + i);
};

ExecutorService executor = Executors.newCachedThreadPool();
executor.execute(hellos);
executor.execute(goodbyes);
}
```

Run the program a few times to see how the outputs are interleaved.

```
Goodbye 1
...
Goodbye 871
Goodbye 872
Hello 806
Goodbye 873
Goodbye 874
Goodbye 875
Goodbye 876
Goodbye 877
Goodbye 878
Goodbye 879
Goodbye 880
Goodbye 881
Hello 807
Goodbye 882
...
Hello 1000
```

Note

You may note that the program waits a bit after the last printout. It terminates when the pooled threads have been idle for a while and the executor service terminates them.

Caution

If concurrent tasks try to read or update a shared value, the result may be unpredictable. We will discuss this issue in detail in Section 10.3, “Thread Safety” (page 341). For now, we will assume that tasks do not share mutable data.

10.1.2 Futures

A Runnable carries out a task, but it doesn't yield a value. If you have a task that computes a result, use the Callable<V> interface instead. Its call method, unlike the run method of the Runnable interface, returns a value of type V:

Click here to view code image

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

As a bonus, the call method can throw arbitrary exceptions which can be relayed to the code that obtains the result.

To execute a Callable, submit it to an executor service:

Click here to view code image

```
ExecutorService executor = Executors.newFixedThreadPool();
Callable<V> task = ...;
Future<V> result = executor.submit(task);
```

When you submit the task, you get a *future*—an object that represents a computation whose result will be available at some future time. The Future interface has the following methods:

Click here to view code image

```
V get() throws InterruptedException, ExecutionException
V get(long timeout, TimeUnit unit)
throws InterruptedException, ExecutionException, TimeoutException;
boolean cancel(boolean mayInterruptIfRunning)
boolean isCancelled()
boolean isDone()
```

The `get` method *blocks* until the result is available or until the timeout has been reached. That is, the thread containing the call does not progress until the method returns normally or throws an exception. If the `call` method yields a value, the `get` method returns that value. If the `call` method throws an exception, the `get` method throws an `ExecutionException` wrapping the thrown exception. If the timeout has been reached, the `get` method throws a `TimeoutException`.

The `cancel` method attempts to cancel the task. If the task isn't already running, it won't be scheduled. Otherwise, if `mayInterruptIfRunning` is `true`, the thread running the task is interrupted.

Note

A task that wants to be interruptible must periodically check for interruption requests. This is required for any tasks that you'd like to cancel when some other subtask has succeeded. See Section 10.8.2, “Thread Interruption” (page 364) for more details on interruption.

A task may need to wait for the result of multiple subtasks. Instead of submitting each subtask separately, you can use the `invokeAll` method, passing a `Collection` of `Callable` instances.

For example, suppose you want to count how often a word occurs in a set of files. For each file, make a `Callable<Integer>` that returns the count for that file. Then submit them all to the executor. When all tasks have completed, you get a list of the futures (all of which are done), and you can total up the answers.

Click here to view code image

```
String word = ...;
Set<Path> paths = ...;
List<Callable<Long>> tasks = new ArrayList<>();
for (Path p : paths) tasks.add(
() -> { return number of occurrences of word in p });
List<Future<Long>> results = executor.invokeAll(tasks);
// This call blocks until all tasks have completed
long total = 0;
for (Future<Long> result : results) total += result.get();
```

There is also a variant of `invokeAll` with a timeout, which cancels all tasks that have not completed when the timeout is reached.

Note

If it bothers you that the calling task blocks until all subtasks are done, you can use an `ExecutorCompletionService`. It returns the futures in the order of completion.

Click here to view code image

```
ExecutorCompletionService service
= new ExecutorCompletionService(executor);
for (Callable<T> task : tasks) service.submit(task);
for (int i = 0; i < tasks.size(); i++) {
Process service.take().get()
Do something else
}
```

The `invokeAny` method is like `invokeAll`, but it returns as soon as any one of the submitted tasks has completed normally, without throwing an exception. It then returns the value of its `Future`. The other tasks are cancelled. This is useful for a search that can conclude as soon as a match has been found. This code snippet locates a file containing a given word:

Click here to view code image

```
String word = ...;
Set<Path> files = ...;
List<Callable<Path>> tasks = new ArrayList<>();
for (Path p : files) tasks.add(
() -> { if (word occurs in p) return p; else throw ... });
Path found = executor.invokeAny(tasks);
```

As you can see, the `ExecutorService` does a lot of work for you. Not only does it map tasks to threads, but it also deals with task results, exceptions, and cancellation.

Note

Java EE provides a `ManagedExecutorService` subclass that is suitable for concurrent tasks in a Java EE environment.

10.2 Asynchronous Computations

In the preceding section, our approach to concurrent computation was to break up a task and then wait until all pieces have completed. But waiting is not always a good idea. In the following sections, you will see how to implement wait-free or *asynchronous* computations.

10.2.1 Completable Futures

When you have a `Future` object, you need to call `get` to obtain the value, blocking until the value is available. The `CompletableFuture` class implements the `Future` interface, and it provides a second mechanism for obtaining the result. You register a *callback* that will be invoked (in some thread) with the result once it is available.

Click here to view code image

```
CompletableFuture<String> f = ...;
f.thenAccept((String s) -> Process the result s);
```

In this way, you can process the result, without blocking, as soon as it is available.

There are a few API methods that return `CompletableFuture` objects. For example, the `HttpClient` class can fetch a web page asynchronously:

Click here to view code image

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder(new
URI(urlString)).GET().build();
CompletableFuture<HttpResponse<String>> f = client.sendAsync(
request, BodyHandler.asString());
```

To run a task asynchronously and obtain a `CompletableFuture`, you don't submit it directly to an executor service. Instead, you call the static method `CompletableFuture.supplyAsync`:

Click here to view code image

```
CompletableFuture<String> f = CompletableFuture.supplyAsync(
() -> { String result; Compute the result; return result; },
executor);
```

If you omit the executor, the task is run on a default executor (namely the executor returned by `ForkJoinPool.commonPool()`).

Note that the first argument of this method is a `Supplier<T>`, not a `Callable<T>`. Both interfaces describe functions with no arguments and a return value of type `T`, but a `Supplier` function cannot throw a checked exception.

A `CompletableFuture` can complete in two ways: either with a result, or with an uncaught exception. In order to handle both cases, use the whenComplete method. The supplied function is called with the result (or null if none) and the exception (or null if none).

Click here to view code image

```
f.whenComplete((s, t) -> {
if (t == null) { Process the result s; }
else { Process the Throwable t; }
});
```

The CompletableFuture is called completable because you can manually set a completion value. (In other concurrency libraries, such an object is called a *promise*). Of course, when you create a CompletableFuture with supplyAsync, the completion value is implicitly set when the task has finished. But setting the result explicitly gives you additional flexibility. For example, two tasks can work simultaneously on computing an answer:

Click here to view code image

```
CompletableFuture<Integer> f = new CompletableFuture<>();
executor.execute(() -> {
int n = workHard(arg);
f.complete(n);
});
executor.execute(() -> {
int n = workSmart(arg);
f.complete(n);
});
```

To instead complete a future with an exception, call

```
Throwable t = ...;
f.completeExceptionally(t);
```

Note

It is safe to call complete or completeExceptionally on the same future in multiple threads. If the future is already completed, these calls have no effect.

The isDone method tells you whether a Future object has been completed (normally or with an exception). In the preceding example, the workHard and workSmart methods can use that information to stop working when the result has been determined by the other method.

Caution

> Unlike a plain `Future`, the computation of a `CompletableFuture` is not interrupted when you invoke its `cancel` method. Canceling simply sets the `Future` object to be completed exceptionally, with a `CancellationException`. In general, this makes sense since a `CompletableFuture` may not have a single thread that is responsible for its completion. However, this restriction also applies to `CompletableFuture` instances returned by methods such as `supplyAsync`, which could in principle be interrupted. See Exercise 27 for a workaround.

10.2.2 Composing Completable Futures

Nonblocking calls are implemented through callbacks. The programmer registers a callback for the action that should occur after a task completes. Of course, if the next action is also asynchronous, the next action after that is in a different callback. Even though the programmer thinks in terms of “first do step 1, then step 2, then step 3,” the program logic can become dispersed in “callback hell.” It gets even worse when you have to add error handling. Suppose step 2 is “the user logs in.” You may need to repeat that step since the user can mistype the credentials. Trying to implement such a control flow in a set of callbacks—or to understand it once it has been implemented—can be quite challenging.

The `CompletableFuture` class solves this problem by providing a mechanism for *composing* asynchronous tasks into a processing pipeline.

For example, suppose we want to extract all links from a web page in order to build a web crawler. Let's say we have a method

Click here to view code image

```
public void CompletableFuture<String> readPage(URI url)
```

that yields the text of a web page when it becomes available. If the method

Click here to view code image

```
public static List<URI> getLinks(String page)
```

yields the URIs in an HTML page, you can schedule it to be called when the page is available:

Click here to view code image

```
CompletableFuture<String> contents = readPage(url);
```

```
CompletableFuture<List<URI>> links =
contents.thenApply(Parser::getLinks);
```

The `thenApply` method doesn't block either. It returns another future. When the first future has completed, its result is fed to the `getLinks` method, and the return value of that method becomes the final result.

With completable futures, you just specify what you want to have done and in which order. It won't all happen right away, of course, but what is important is that all the code is in one place.

Conceptually, `CompletableFuture` is a simple API, but there are many variants of methods for composing completable futures. Let us first look at those that deal with a single future (see Table 10-1). (For each method shown, there are also two `Async` variants that I don't show. One of them uses a shared `ForkJoinPool`, and the other has an `Executor` parameter.) In the table, I use a shorthand notation for the ponderous functional interfaces, writing `T -> U` instead of `Function<? super T, U>`. These aren't actual Java types, of course.

You have already seen the `thenApply` method. Suppose `f` is a function that receives values of type `T` and returns values of type `U`. The calls

Click here to view code image

```
CompletableFuture<U> future.thenApply(f);
CompletableFuture<U> future.thenApplyAsync(f);
```

return a future that applies the function `f` to the result of `future` when it is available. The second call runs `f` in yet another thread.

The `thenCompose` method, instead of taking a function mapping the type `T` to the type `U`, receives a function mapping `T` to `CompletableFuture<U>`. That sounds rather abstract, but it can be quite natural. Consider the action of reading a web page from a given URL. Instead of supplying a method

Click here to view code image

```
public String blockingReadPage(URI url)
```

it is more elegant to have that method return a future:

Click here to view code image

```
public CompletableFuture<String> readPage(URI url)
```

Now, suppose we have another method that gets the URL from user input, perhaps from a dialog that won't reveal the answer until the user has clicked the

OK button. That, too, is an event in the future:

Click here to view code image

```
public CompletableFuture<URI> getURLInput(String prompt)
```

Here we have two functions `T -> CompletableFuture<U>` and `U -> CompletableFuture<V>`. Clearly, they compose to a function `T -> CompletableFuture<V>` if the second function is called when the first one has completed. That is exactly what `thenCompose` does.

In the preceding section, you saw the `whenComplete` method for handling exceptions. There is also a `handle` method that requires a function processing the result or exception and computing a new result. In many cases, it is simpler to call the `exceptionally` method instead:

Click here to view code image

```
CompletableFuture<String> contents = readPage(url)
.exceptionally(t -> { Log t; return emptyPage; });
```

The supplied handler is only called if an exception occurred, and it produces a result to be used in the processing pipeline. If no exception occurred, the original result is used.

The methods in Table 10-1 with `void` result are normally used at the end of a processing pipeline.

Table 10-1 Adding an Action to a `CompletableFuture<T>` Object

Method	Parameter	Description
`thenApply`	`T -> U`	Apply a function to the result.
`thenAccept`	`T -> void`	Like `thenApply`, but with `void` result.
`thenCompose`	`T -> CompletableFuture<U>`	Invoke the function on the result and execute the returned future.
`handle`	`(T, Throwable) -> U`	Process the result or error and yield a new result.
`whenComplete`	`(T, Throwable) -> void`	Like `handle`, but with `void` result.
`exceptionally`	`Throwable -> T`	Turn the error into a default result.

`thenRun`	`Runnable`	Execute the `Runnable` with `void` result.

Now let us turn to methods that combine multiple futures (see Table 10-2).

Table 10-2 Combining Multiple Composition Objects

Method	**Parameters**	**Description**
`thenCombine`	`CompletableFuture<U>`, `(T, U) -> V`	Execute both and combine the results with the given function.
`thenAcceptBoth`	`CompletableFuture<U>`, `(T, U) -> void`	Like `thenCombine`, but with `void` result.
`runAfterBoth`	`CompletableFuture<?>`, `Runnable`	Execute the runnable after both complete.
`applyToEither`	`CompletableFuture<T>`, `T -> V`	When a result is available from one or the other, pass it to the given function.
`acceptEither`	`CompletableFuture<T>`, `T -> void`	Like `applyToEither`, but with `void` result.
`runAfterEither`	`CompletableFuture<?>`, `Runnable`	Execute the runnable after one or the other completes.
`static allOf`	`CompletableFuture<?>...`	Complete with `void` result after all given futures complete.
`static anyOf`	`CompletableFuture<?>...`	Complete after any of the given futures completes and yield its result.

The first three methods run a `CompletableFuture<T>` and a `CompletableFuture<U>` action concurrently and combine the results.

The next three methods run two `CompletableFuture<T>` actions concurrently. As soon as one of them finishes, its result is passed on, and the other result is ignored.

Finally, the static `allOf` and `anyOf` methods take a variable number of completable futures and yield a `CompletableFuture<Void>` that completes when all of them, or any one of them, completes. The `allOf` method does not yield a result. The `anyOf` method does *not* terminate the remaining tasks. Exercises 28 and 29 show useful improvements of these two methods.

Note

> Technically speaking, the methods in this section accept parameters of type `CompletionStage`, not `CompletableFuture`. The `CompletionStage` interface describes how to compose asynchronous computations, whereas the `Future` interface focuses on the result of a computation. A `CompletableFuture` is both a `CompletionStage` and a `Future`.

10.2.3 Long-Running Tasks in User-Interface Callbacks

One of the reasons to use threads is to make your programs more responsive. This is particularly important in an application with a user interface. When your program needs to do something time-consuming, you cannot do the work in the user-interface thread, or the user interface will freeze. Instead, fire up another worker thread.

For example, if you want to read a web page when the user clicks a button, don't do this:

Click here to view code image

```
Button read = new Button("Read");
read.setOnAction(event -> { // Bad—long-running action is executed on UI thread
Scanner in = new Scanner(url.openStream());
while (in.hasNextLine()) {
String line = in.nextLine();
...
}
});
```

Instead, do the work in a separate thread.

Click here to view code image

```
read.setOnAction(event -> { // Good—long-running action in separate thread
Runnable task = () -> {
Scanner in = new Scanner(url.openStream());
while (in.hasNextLine()) {
String line = in.nextLine();
...
}
}
executor.execute(task);
});
```

However, you cannot directly update the user interface from the thread that executes the long-running task. User interfaces such as JavaFX, Swing, or Android are not threadsafe. You cannot manipulate user-interface elements from multiple threads, or they risk becoming corrupted. In fact, JavaFX and Android check for this, and throw an exception if you try to access the user interface from a thread other than the UI thread.

Therefore, you need to schedule any UI updates to happen on the UI thread. Each user-interface library provides some mechanism to schedule a `Runnable` for execution on the UI thread. For example, in JavaFX, you can use

Click here to view code image

```
Platform.runLater(() ->
message.appendText(line + "\n"));
```

Note

It is tedious to implement lengthy operations while giving users feedback on the progress, so user-interface libraries usually provide some kind of helper class for managing the details, such as `SwingWorker` in Swing and `AsyncTask` in Android. You specify actions for the long-running task (which is run on a separate thread), as well as progress updates and the final disposition (which are run on the UI thread).

The `Task` class in JavaFX takes a slightly different approach to progress updates. The class provides methods to update task properties (a message, completion percentage, and result value) in the long-running thread. You bind the properties to user-interface elements, which are then updated in the UI thread.

10.3 Thread Safety

Many programmers initially think that concurrent programming is pretty easy. You just divide your work into tasks, and that's it. What could possibly go wrong?

In the following sections, I show you what can go wrong, and give a high-level overview of what you can do about it.

10.3.1 Visibility

Even operations as simple as writing and reading a variable can be incredibly complicated with modern processors. Consider this example:

Click here to view code image

```
private static boolean done = false;

public static void main(String[] args) {
Runnable hellos = () -> {
for (int i = 1; i <= 1000; i++)
System.out.println("Hello " + i);
done = true;
};
Runnable goodbye = () -> {
int i = 1;
while (!done) i++;
System.out.println("Goodbye " + i);
};
Executor executor = Executors.newCachedThreadPool();
executor.execute(hellos);
executor.execute(goodbye);
}
```

The first task prints “`Hello`” a thousand times, and then sets `done` to `true`. The second task waits for `done` to become `true`, and then prints “`Goodbye`” once, incrementing a counter while it is waiting for that happy moment.

You'd expect the output to be something like

```
Hello 1
...
Hello 1000
Goodbye 501249
```

When I run this program on my laptop, the program prints up to “`Hello 1000`” and never terminates. The effect of

```
done = true;
```

may not be *visible* to the thread running the second task.

Why wouldn't it be visible? Modern compilers, virtual machines, and processors perform many optimizations. These optimizations assume that the code is sequential unless explicitly told otherwise.

One optimization is caching of memory locations. We think of a memory location such as `done` as bits somewhere in the transistors of a RAM chip. But

RAM chips are slow—many times slower than modern processors. Therefore, a processor tries to hold the data that it needs in registers or an onboard memory cache, and eventually writes changes back to memory. This caching is simply indispensable for processor performance. There are operations for synchronizing cached copies, but they have a significant performance cost and are only issued when requested.

Another optimization is instruction reordering. The compiler, the virtual machine, and the processor are allowed to change the order of instructions to speed up operations, provided it does not change the sequential semantics of the program.

For example, consider a computation

Click here to view code image

```
x = Something not involving y;
y = Something not involving x;
z = x + y;
```

The first two steps must occur before the third, but they can occur in either order. A processor can (and often will) run the first two steps concurrently, or swap the order if the inputs to the second step are more quickly available.

In our case, the loop

```
while (!done) i++;
```

can be reordered as

```
if (!done) while (true) i++;
```

since the loop body does not change the value of `done`.

By default, optimizations assume that there are no concurrent memory accesses. If there are, the virtual machine needs to know, so that it can then emit processor instructions that inhibit improper reorderings.

There are several ways of ensuring that an update to a variable is visible. Here is a summary:

1. The value of a `final` variable is visible after initialization.
2. The initial value of a `static` variable is visible after static initialization.
3. Changes to a `volatile` variable are visible.
4. Changes that happen before releasing a lock are visible to anyone acquiring the same lock (see Section 10.7.1, “Locks,” page 357).

In our case, the problem goes away if you declare the shared variable `done` with the volatile modifier:

Click here to view code image

```
private static volatile boolean done;
```

Then the compiler generates instructions that cause the virtual machine to issue processor commands for cache synchronization. As a result, any change to done in one task becomes visible to the other tasks.

The volatile modifier happens to suffice to solve this particular problem. But as you will see in the next section, declaring shared variables as volatile is not a general solution.

Tip

It is an excellent idea to declare any field that does not change after initialization as final. Then you never have to worry about its visibility.

10.3.2 Race Conditions

Suppose multiple concurrent tasks update a shared integer counter.

Click here to view code image

```
private static volatile int count = 0;
...
count++; // Task 1
...
count++; // Task 2
...
```

The variable has been declared as volatile, so the updates are visible. But that is not enough.

The update count++ actually means

```
register = count + 1;
count = register;
```

When these computations are interleaved, the wrong value can be stored back into the count variable. In the parlance of concurrency, we say that the increment operation is not *atomic*. Consider this scenario:

Click here to view code image

```
int count = 0; // Initial value
```

```
register1 = count + 1; // Thread 1 computes count + 1
... // Thread 1 is preempted
register2 = count + 1; // Thread 2 computes count + 1
count = register2; // Thread 2 stores 1 in count
... // Thread 1 is running again
count = register1; // Thread 1 stores 1 in count
```

Now `count` is `1`, not `2`. This kind of error is called a *race condition* because it depends on which thread wins the “race” for updating the shared variable.

Does this problem really happen? It certainly does. Run the demo program of the companion code. It has 100 threads, each incrementing the counter 1,000 times and printing the result.

Click here to view code image

```
for (int i = 1; i <= 100; i++) {
int taskId = i;
Runnable task = () -> {
for (int k = 1; k <= 1000; k++)
count++;
System.out.println(taskId + ": " + count);
};
executor.execute(task);
}
```

The output usually starts harmlessly enough as something like

```
1: 1000
3: 2000
2: 3000
6: 4000
```

After a while, it looks a bit scary:

```
72: 58196
68: 59196
73: 61196
71: 60196
69: 62196
```

But that might just be because some threads were paused at inopportune moments. What matters is what happens with the task that finished last. Did it bring up the counter to 100,000?

I ran the program dozens of times on my multi-core laptop, and it fails every time. Years ago, when personal computers had a single CPU, race conditions were more difficult to observe, and programmers did not notice such dramatic failures often. But it doesn't matter whether a wrong value is computed within seconds or hours.

This example looks at the simple case of a shared counter in a toy program. Exercise 17 shows the same problem in a realistic example. But it's not just counters. Race conditions are a problem whenever shared variables are mutated. For example, when adding a value to the head of a queue, the insertion code might look like this:

```
Node n = new Node();
if (head == null) head = n;
else tail.next = n;
tail = n;
tail.value = newValue;
```

Lots of things can go wrong if this sequence of instructions is paused at an unfortunate time and another task gets control, accessing the queue while it is in an inconsistent state.

Work through Exercise 21 to get a feel for how a data structure can get corrupted by concurrent mutation.

We need to ensure that the entire sequence of operation is carried out together. Such an instruction sequence is called a *critical section*. You can use a *lock* to protect critical sections and make critical sequences of operation atomic. You will learn how to program with locks in Section 10.7.1, “Locks” (page 357).

While it is straightforward to use locks for protecting critical sections, locks are not a general solution for solving all concurrency problems. They are difficult to use properly, and it is easy to make mistakes that severely degrade performance or even cause “deadlock.”

10.3.3 Strategies for Safe Concurrency

In languages such as C and C++, programmers need to manually allocate and deallocate memory. That sounds dangerous—and it is. Many programmers have spent countless miserable hours chasing memory allocation bugs. In Java, there is a garbage collector, and few Java programmers need to worry about memory management.

Unfortunately, there is no equivalent mechanism for shared data access in a concurrent program. The best you can do is to follow a set of guidelines to manage the inherent dangers.

A highly effective strategy is *confinement*. Just say no when it comes to sharing data among tasks. For example, when your tasks need to count something, give each of them a private counter instead of updating a shared counter. When the tasks are done, they can hand off their results to another task that combines them.

Another good strategy is *immutability*. It is safe to share immutable objects. For example, instead of adding results to a shared collection, a task can generate an immutable collection of results. Another task combines the results into another immutable data structure. The idea is simple, but there are a few things to watch out for—see Section 10.3.4, "Immutable Classes" (page 347).

The third strategy is *locking*. By granting only one task at a time access to a data structure, one can keep it from being damaged. In Section 10.5, "Threadsafe Data Structures" (page 350), you will see data structures provided by the Java concurrency library that are safe to use concurrently. Section 10.7.1, "Locks" (page 357) shows you how locking works, and how experts build these data structures.

Locking is error-prone, and it can be expensive since it reduces opportunities for concurrent execution. For example, if you have lots of tasks contributing results to a shared hash table, and the table is locked for each update, then that is a real bottleneck. If most tasks have to wait their turn, they aren't doing useful work. Sometimes it is possible to *partition* data so that different pieces can be accessed concurrently. Several data structures in the Java concurrency library use partitioning, as do the parallel algorithms in the streams library. Don't try this at home! It is really hard to get it right. Instead, use the data structures and algorithms from the Java library.

10.3.4 Immutable Classes

A class is immutable when its instances, once constructed, cannot change. It sounds at first as if you can't do much with them, but that isn't true. The ubiquitous `String` class is immutable, as are the classes in the date and time library (see Chapter 12). Each date instance is immutable, but you can obtain new dates, such as the one that comes a day after a given one.

Or consider a set for collecting results. You could use a mutable `HashSet` and update it like this:

```
results.addAll(newResults);
```

But that is clearly dangerous.

An immutable set always creates new sets. You would update the results somewhat like this:

Click here to view code image

```
results = results.union(newResults);
```

There is still mutation, but it is much easier to control what happens to one variable than to a hash set with many methods.

It is not difficult to implement immutable classes, but you should pay attention to these issues:

1. Don't change the object state after construction. Be sure to declare instance variables `final`. There is no reason not to, and you gain an important advantage: the virtual machine ensures that a `final` instance variable is visible after construction (Section 10.3.1, “Visibility,” page 342).
2. Of course, none of the methods can be mutators. You should make them `final`, or better, declare the class `final`, so that mutators cannot be added in subclasses.
3. Don't leak state that can be mutated externally. None of your (non-`private`) methods can return a reference to any innards that could be used for mutation, such as an internal array or collection. When one of your methods calls a method of another class, it must not pass any such references either, since the called method might otherwise use them for mutation. Instead, pass a copy.
4. Conversely, don't store any reference to a mutable object that the constructor receives. Instead, make a copy.
5. Don't let the `this` reference escape in a constructor. When you call another method, you know not to pass any internal references, but what about `this`? That's perfectly safe after construction, but if you reveal `this` in the constructor, someone could observe the object in an incomplete state. Also beware of constructors giving out inner class references that contain a hidden `this` reference. Naturally, these situations are quite rare.

10.4 Parallel Algorithms

Before starting to parallelize your computations, you should check if the Java library has done this for you. The stream library or the `Arrays` class may already do what you need.

10.4.1 Parallel Streams

The stream library can automatically parallelize operations on large data sets. For example, if `coll` is a large collection of strings, and you want to find how many of them start with the letter A, call

Click here to view code image

```
long result = coll.parallelStream().filter(s ->
s.startsWith("A")).count();
```

The `parallelStream` method yields a parallel stream. The stream is broken up into segments. The filtering and counting is done on each segment, and the results are combined. You don't need to worry about the details.

Caution

When you use parallel streams with lambdas (for example, as the argument to `filter` and `map` in the preceding examples), be sure to stay away from unsafe mutation of shared objects.

For parallel streams to work well, a number of conditions need to be fulfilled:

- There needs to be enough data. There is a substantial overhead for parallel streams that is only repaid for large data sets.
- The data should be in memory. It would be inefficient to have to wait for the data to arrive.
- The stream should be efficiently splittable into subregions. A stream backed by an array or a balanced binary tree works well, but a linked list or the result of `Stream.iterate` does not.
- The stream operations should do a substantial amount of work. If the total work load is not large, it does not make sense to pay for the cost of setting up the concurrent computation.
- The stream operations should not block.

In other words, don’t turn all your streams into parallel streams. Use parallel streams only when you do a substantial amount of sustained computational work on data that is already in memory.

10.4.2 Parallel Array Operations

The `Arrays` class has a number of parallelized operations. Just as with the parallel stream operations of the preceding sections, the operations break the array into sections, work on them concurrently, and combine the results.

The static `Arrays.parallelSetAll` method fills an array with values computed by a function. The function receives the element index and computes the value at that location.

Click here to view code image

```
Arrays.parallelSetAll(values, i -> i % 10);
// Fills values with 0 1 2 3 4 5 6 7 8 9 0 1 2 ...
```

Clearly, this operation benefits from being parallelized. There are versions for all primitive type arrays and for object arrays.

The `parallelSort` method can sort an array of primitive values or objects. For example,

Click here to view code image

```
Arrays.parallelSort(words, Comparator.comparing(String::length));
```

With all methods, you can supply the bounds of a range, such as

Click here to view code image

```
Arrays.parallelSort(values, values.length / 2, values.length); // Sort
the upper half
```

Note

At first glance, it seems a bit odd that these methods have `parallel` in their names—the user shouldn’t care how the setting or sorting happens. However, the API designers wanted to make it clear that the operations are parallelized. That way, users are on notice to avoid generator or comparison functions with side effects.

Finally, there is a `parallelPrefix` that is rather specialized—Exercise 4 gives a simple example.

For other parallel operations on arrays, turn the arrays into parallel streams. For example, to compute the sum of a long array of integers, call

Click here to view code image

```
long sum = IntStream.of(values).parallel().sum();
```

10.5 Threadsafe Data Structures

If multiple threads concurrently modify a data structure, such as a queue or hash table, it is easy to damage the internals of the data structure. For example, one thread may begin to insert a new element. Suppose it is preempted in the middle of rerouting links, and another thread starts traversing the same location. The second thread may follow invalid links and create havoc, perhaps throwing exceptions or even getting trapped in an infinite loop.

As you will see in Section 10.7.1, “Locks” (page 357), you can use locks to ensure that only one thread can access the data structure at a given point in time, blocking any others. But you can do better than that. The collections in the `java.util.concurrent` package have been cleverly implemented so that multiple threads can access them without blocking each other, provided they access different parts.

Note

These collections yield *weakly consistent* iterators. That means that the iterators present elements appearing at onset of iteration, but may or may not reflect some or all of the modifications that were made after they were constructed. However, such an iterator will not throw a `ConcurrentModificationException`.

In contrast, an iterator of a collection in the `java.util` package throws a `ConcurrentModificationException` when the collection has been modified after construction of the iterator.

10.5.1 Concurrent Hash Maps

A `ConcurrentHashMap` is, first of all, a hash map whose operations are threadsafe. No matter how many threads operate on the map at the same time, the internals are not corrupted. Of course, some threads may be temporarily blocked, but the map can efficiently support a large number of concurrent readers and a certain number of concurrent writers.

But that is not enough. Suppose we want to use a map to count how often certain features are observed. As an example, suppose multiple threads encounter words, and we want to count their frequencies. Obviously, the following code for updating a count is not threadsafe:

Click here to view code image

```
ConcurrentHashMap<String, Long> map = new ConcurrentHashMap<>();
...
Long oldValue = map.get(word);
Long newValue = oldValue == null ? 1 : oldValue + 1;
map.put(word, newValue); // Error—might not replace oldValue
```

Another thread might be updating the exact same count at the same time.

To update a value safely, use the `compute` method. It is called with a key and a function to compute the new value. That function receives the key and the associated value, or `null` if there is none, and computes the new value. For example, here is how we can update a count:

Click here to view code image

```
map.compute(word, (k, v) -> v == null ? 1 : v + 1);
```

The `compute` method is *atomic*—no other thread can mutate the map entry while the computation is in progress.

There are also variants `computeIfPresent` and `computeIfAbsent` that only compute a new value when there is already an old one, or when there isn't yet one.

Another atomic operation is `putIfAbsent`. A counter might be initialized as

```
map.putIfAbsent(word, 0L);
```

You often need to do something special when a key is added for the first time. The `merge` method makes this particularly convenient. It has a parameter for the initial value that is used when the key is not yet present. Otherwise, the function that you supplied is called, combining the existing value and the initial value. (Unlike `compute`, the function does not process the key.)

Click here to view code image

```
map.merge(word, 1L, (existingValue, newValue) -> existingValue +
newValue);
```

or simply,

Click here to view code image

```
map.merge(word, 1L, Long::sum);
```

Of course, the functions passed to `compute` and `merge` should complete quickly, and they should not attempt to mutate the map.

Note

There are methods that atomically remove or replace an entry if it is currently equal to an existing one. Before the `compute` method was available, people would write code like this for incrementing a count:

Click here to view code image

```
do {
oldValue = map.get(word);
newValue = oldValue + 1;
} while (!map.replace(word, oldValue, newValue));
```

Note

There are several *bulk operations* for searching, transforming, or visiting a `ConcurrentHashMap`. They operate on a snapshot of the data and can safely execute even while other threads operate on the map. In the API documentation, look for the operations whose names start with `search`, `reduce`, and `forEach`. There are variants that operate on the keys, values, and entries. The `reduce` methods have specializations for `int`-, `long`-, and `double`-valued reduction functions.

10.5.2 Blocking Queues

One commonly used tool for coordinating work between tasks is a *blocking queue*. Producer tasks insert items into the queue, and consumer tasks retrieve them. The queue lets you safely hand over data from one task to another.

When you try to add an element and the queue is currently full, or you try to remove an element when the queue is empty, the operation blocks. In this way, the queue balances the workload. If the producer tasks run slower than the consumer tasks, the consumers block while waiting for the results. If the producers run faster, the queue fills up until the consumers catch up.

Table 10-3 shows the methods for blocking queues. The blocking queue methods fall into three categories that differ by the action they perform when the queue is full or empty. In addition to the blocking methods, there are methods that throw an exception when they don't succeed, and methods that return with a failure indicator instead of throwing an exception if they cannot carry out their tasks.

Table 10-3 Blocking Queue Operations

Method	Normal Action	Error Action
`put`	Adds an element to the tail	Blocks if the queue is full
`take`	Removes and returns the head element	Blocks if the queue is empty
`add`	Adds an element to the tail	Throws an `IllegalStateException` if the queue is full
	Removes and returns the	Throws a `NoSuchElementException`

`remove`	head element	if the queue is empty
`element`	Returns the head element	Throws a `NoSuchElementException` if the queue is empty
`offer`	Adds an element and returns `true`	Returns `false` if the queue is full
`poll`	Removes and returns the head element	Returns `null` if the queue is empty
`peek`	Returns the head element	Returns `null` if the queue is empty

Note

The `poll` and `peek` methods return `null` to indicate failure. Therefore, it is illegal to insert null values into these queues.

There are also variants of the `offer` and `poll` methods with a timeout. For example, the call

Click here to view code image

```
boolean success = q.offer(x, 100, TimeUnit.MILLISECONDS);
```

tries for 100 milliseconds to insert an element to the tail of the queue. If it succeeds, it returns `true`; otherwise, it returns `false` when it times out. Similarly, the call

Click here to view code image

```
Object head = q.poll(100, TimeUnit.MILLISECONDS)
```

tries for 100 milliseconds to remove the head of the queue. If it succeeds, it returns the head; otherwise, it returns `null` when it times out.

The `java.util.concurrent` package supplies several variations of blocking queues. A `LinkedBlockingQueue` is based on a linked list, and an `ArrayBlockingQueue` uses a circular array.

Exercise 11 shows how to use blocking queues for analyzing files in a directory. One thread walks the file tree and inserts files into a queue. Several threads remove the files and search them. In this application, it is likely that the producer quickly fills up the queue with files and blocks until the consumers can catch up.

A common challenge with such a design is stopping the consumers. A consumer cannot simply quit when the queue is empty. After all, the producer might not yet have started, or it may have fallen behind. If there is a single producer, it can add a “last item” indicator to the queue, similar to a dummy suitcase with a label “last bag” in a baggage claim belt.

10.5.3 Other Threadsafe Data Structures

Just like you can choose between hash maps and tree maps in the `java.util` package, there is a concurrent map that is based on comparing keys, called `ConcurrentSkipListMap`. Use it if you need to traverse the keys in sorted order, or if you need one of the added methods in the `NavigableMap` interface (see Chapter 7). Similarly, there is a `ConcurrentSkipListSet`.

The `CopyOnWriteArrayList` and `CopyOnWriteArraySet` are threadsafe collections in which all mutators make a copy of the underlying array. This arrangement is useful if the threads that iterate over the collection greatly outnumber the threads that mutate it. When you construct an iterator, it contains a reference to the current array. If the array is later mutated, the iterator still has the old array, but the collection’s array is replaced. As a consequence, the older iterator has a consistent (but potentially outdated) view that it can access without any synchronization expense.

Suppose you want a large, threadsafe set instead of a map. There is no `ConcurrentHashSet` class, and you know better than trying to create your own. Of course, you can use a `ConcurrentHashMap` with bogus values, but that gives you a map, not a set, and you can't apply operations of the `Set` interface.

The static `newKeySet` method yields a `Set<K>` that is actually a wrapper around a `ConcurrentHashMap<K, Boolean>`. (All map values are `Boolean.TRUE`, but you don't actually care since you just use it as a set.)

Click here to view code image

```
Set<String> words = ConcurrentHashMap.newKeySet();
```

If you have an existing map, the `keySet` method yields the set of keys. That set is mutable. If you remove the set's elements, the keys (and their values) are removed from the map. But it doesn't make sense to add elements to the key set, because there would be no corresponding values to add. You can use a second `keySet` method, with a default value used when adding elements to the set:

Click here to view code image

```
Set<String> words = map.keySet(1L);
words.add("Java");
```

If "Java" wasn't already present in words, it now has a value of one.

10.6 Atomic Counters and Accumulators

If multiple threads update a shared counter, you need to make sure that this is done in a threadsafe way. There are a number of classes in the java.util.concurrent.atomic package that use safe and efficient machine-level instructions to guarantee atomicity of operations on integers, long and boolean values, object references, and arrays thereof. Using these classes correctly requires considerable expertise. However, atomic counters and accumulators are convenient for application-level programming.

For example, you can safely generate a sequence of numbers like this:

Click here to view code image

```
public static AtomicLong nextNumber = new AtomicLong();
// In some thread...
long id = nextNumber.incrementAndGet();
```

The incrementAndGet method atomically increments the AtomicLong and returns the post-increment value. That is, the operations of getting the value, adding 1, setting it, and producing the new value cannot be interrupted. It is guaranteed that the correct value is computed and returned, even if multiple threads access the same instance concurrently.

There are methods for atomically setting, adding, and subtracting values, but suppose you want to make a more complex update. One way is to use the updateAndGet method. For example, suppose you want to keep track of the largest value that is observed by different threads. The following won't work:

Click here to view code image

```
public static AtomicLong largest = new AtomicLong();
// In some thread...
largest.set(Math.max(largest.get(), observed)); // Error—race condition!
```

This update is not atomic. Instead, call updateAndGet with a lambda expression for updating the variable. In our example, we can call

Click here to view code image

```
largest.updateAndGet(x -> Math.max(x, observed));
```

or

Click here to view code image

```
largest.accumulateAndGet(observed, Math::max);
```

The `accumulateAndGet` method takes a binary operator that is used to combine the atomic value and the supplied argument.

There are also methods `getAndUpdate` and `getAndAccumulate` that return the old value.

Note

These methods are also provided for the classes:

Click here to view code image

```
AtomicInteger AtomicLongFieldUpdater
AtomicIntegerArray AtomicReference
AtomicIntegerFieldUpdater AtomicReferenceArray
AtomicLongArray AtomicReferenceFieldUpdater
```

When you have a very large number of threads accessing the same atomic values, performance suffers because updates are carried out *optimistically*. That is, the operation computes a new value from a given old value, then does the replacement provided the old value is still the current one, or retries if it is not. Under heavy contention, updates require too many retries.

The classes `LongAdder` and `LongAccumulator` solve this problem for certain common updates. A `LongAdder` is composed of multiple variables whose collective sum is the current value. Multiple threads can update different summands, and new summands are automatically provided when the number of threads increases. This is efficient in the common situation where the value of the sum is not needed until after all work has been done. The performance improvement can be substantial—see Exercise 9.

If you anticipate high contention, you should simply use a `LongAdder` instead of an `AtomicLong`. The method names are slightly different. Call `increment` to increment a counter or `add` to add a quantity, and `sum` to retrieve the total.

Click here to view code image

```
final LongAdder count = new LongAdder();
for (...)
executor.execute(() -> {
while (...) {
```

```
...
if (...) count.increment();
}
});
...
long total = count.sum();
```

Note

Of course, the `increment` method does *not* return the old value. Doing that would undo the efficiency gain of splitting the sum into multiple summands.

The `LongAccumulator` generalizes this idea to an arbitrary accumulation operation. In the constructor, you provide the operation as well as its neutral element. To incorporate new values, call `accumulate`. Call `get` to obtain the current value.

Click here to view code image

```
LongAccumulator accumulator = new LongAccumulator(Long::sum, 0);
// In some tasks...
accumulator.accumulate(value);
// When all work is done
long sum = accumulator.get();
```

Internally, the accumulator has variables a_1, a_2, …, a_n. Each variable is initialized with the neutral element (`0` in our example).

When `accumulate` is called with value *v*, then one of them is atomically updated as $a_i = a_i \; op \; v$, where *op* is the accumulation operation written in infix form. In our example, a call to `accumulate` computes $a_i = a_i + v$ for some *i*.

The result of `get` is $a_1 \; op \; a_2 \; op \; \ldots \; op \; a_n$. In our example, that is the sum of the accumulators, $a_1 + a_2 + \ldots + a_n$.

If you choose a different operation, you can compute maximum or minimum (see Exercise 10). In general, the operation must be associative and commutative. That means that the final result must be independent of the order in which the intermediate values were combined.

There are also `DoubleAdder` and `DoubleAccumulator` that work in the same way, except with `double` values.

Tip

If you use a hash map of `LongAdder`, you can use the following idiom to increment the adder for a key:

Click here to view code image

```
ConcurrentHashMap<String,LongAdder> counts = ...;
counts.computeIfAbsent(key, k -> new LongAdder()).increment();
```

When the count for `key` is incremented the first time, a new adder is set.

10.7 Locks and Conditions

Now you have seen several tools that application programmers can safely use for structuring concurrent applications. You may be curious how one would build a threadsafe counter or blocking queue. The following sections show you how it is done, so that you gain some understanding of the costs and complexities.

10.7.1 Locks

To avoid the corruption of shared variables, one needs to ensure that only one thread at a time can compute and set the new values. Code that must be executed in its entirety, without interruption, is called a *critical section*. One can use a *lock* to implement a critical section:

Click here to view code image

```
Lock countLock = new ReentrantLock(); // Shared among multiple threads
int count; // Shared among multiple threads
...
countLock.lock();
try {
count++; // Critical section
} finally {
countLock.unlock(); // Make sure the lock is unlocked
}
```

Note

In this section, I use the `ReentrantLock` class to explain how locking works. As you will see in the next section, there is no requirement to use explicit locks since there are “implicit” locks that are used by the `synchronized` keyword. But it is easier to understand what goes on under the hood by looking at explicit locks.

The first thread to execute the `lock` method locks the `countLock` object and then proceeds into the critical section. If another thread tries to call `lock` on the same object, it is blocked until the first thread executes the call to `unlock`. In this way, it is guaranteed that only one thread at a time can execute the critical section.

Note that, by placing the `unlock` method into a `finally` clause, the lock is released if any exception happens in the critical section. Otherwise, the lock would be permanently locked, and no other thread would be able to proceed past it. This would clearly be very bad. Of course, in this case, the critical section can't throw an exception since it only executes an integer increment. But it is a common idiom to use the `try`/`finally` statement anyway, in case more code gets added later.

At first glance, it seems simple enough to use locks for protecting critical sections. However, the devil is in the details. Experience has shown that many programmers have difficulty writing correct code with locks. They might use the wrong locks, or create situations that *deadlock* when no thread can make progress because all of them wait for a lock.

For that reason, application programmers should use locks as a matter of last resort. First try to avoid sharing, by using immutable data or handing off mutable data from one thread to another. If you must share, use prebuilt threadsafe structures such as a `ConcurrentHashMap` or a `LongAdder`. Still, it is useful to know about locks so you can understand how such data structures can be implemented.

10.7.2 The `synchronized` Keyword

In the preceding section, I showed you how to use a `ReentrantLock` to implement a critical section. You don't have to use an explicit lock because in Java, *every object* has an *intrinsic lock*. To understand intrinsic locks, however, it helps to have seen explicit locks first.

The `synchronized` keyword is used to lock the intrinsic lock. It can occur in two forms. You can lock a block:

```
synchronized (obj) {
Critical section
}
```

This essentially means

Click here to view code image

```
obj.intrinsicLock.lock();
try {
Critical section
} finally {
obj.intrinsicLock.unlock();
}
```

An object does not actually have a field that is an intrinsic lock. The code is just meant to illustrate what goes on when you use the `synchronized` keyword.

You can also declare a method as `synchronized`. Then its body is locked on the receiver parameter `this`. That is,

Click here to view code image

```
public synchronized void method() {
Body
}
```

is the equivalent of

Click here to view code image

```
public void method() {
this.intrinsicLock.lock();
try {
Body
} finally {
this.intrinsicLock.unlock();
}
}
```

For example, a counter can simply be declared as

Click here to view code image

```
public class Counter {
private int value;
public synchronized int increment() {
value++;
return value;
}
}
```

By using the intrinsic lock of the `Counter` instance, there is no need to come up with an explicit lock.

As you can see, using the `synchronized` keyword yields code that is quite concise. Of course, to understand this code, you have to know that each object has an intrinsic lock.

Note

> There is more to locks than atomicity. Locks also guarantee visibility. For example, consider the `done` variable that gave us so much grief in Section 10.3.1, “Visibility” (page 342). If you use a lock for both writing and reading the variable, then you are assured that the caller of `get` sees any update to the variable through a call by `set`.

Click here to view code image

```
public class Flag {
private boolean done;
public synchronized void set() { done = true; }
public synchronized boolean get() { return done; }
}
```

Synchronized methods were inspired by the *monitor* concept that was pioneered by Per Brinch Hansen and Tony Hoare in the 1970s. A monitor is essentially a class in which all instance variables are private and all methods are protected by a private lock.

In Java, it is possible to have public instance variables and to mix synchronized and unsynchronized methods. More problematically, the intrinsic lock is publicly accessible.

Many programmers find this confusing. For example, Java 1.0 has a `Hashtable` class with synchronized methods for mutating the table. To safely iterate over such a table, you can acquire the lock like this:

Click here to view code image

```
synchronized (table) {
for (K key : table.keySet()) ...
}
```

Here, `table` denotes both the hash table and the lock that its methods use. This is a common source of misunderstandings—see Exercise 22.

10.7.3 Waiting on Conditions

Consider a simple `Queue` class with methods for adding and removing objects. Synchronizing the methods ensures that these operations are atomic.

Click here to view code image

```
public class Queue {
class Node { Object value; Node next; };
private Node head;
private Node tail;

public synchronized void add(Object newValue) {
Node n = new Node();
if (head == null) head = n;
else tail.next = n;
tail = n;
tail.value = newValue;
}

public synchronized Object remove() {
if (head == null) return null;
Node n = head;
head = n.next;
return n.value;
}
}
```

Now suppose we want to turn the `remove` method into a method `take` that blocks if the queue is empty.

The check for emptiness must come inside the synchronized method because otherwise the inquiry would be meaningless—another thread might have emptied the queue in the meantime.

Click here to view code image

```
public synchronized Object take() {
if (head == null) ... // Now what?
Node n = head;
head = n.next;
return n.value;
}
```

But what should happen if the queue is empty? No other thread can add elements while the current thread holds the lock. This is where the `wait` method comes in.

If the `take` method finds that it cannot proceed, it calls the `wait` method:

Click here to view code image

```
public synchronized Object take() throws InterruptedException {
while (head == null) wait();
...
}
```

The current thread is now deactivated and gives up the lock. This lets in another thread that can, we hope, add elements to the queue. This is called *waiting on a condition*.

Note that the `wait` method is a method of the `Object` class. It relates to the lock that is associated with the object.

There is an essential difference between a thread that is blocking to acquire a lock and a thread that has called `wait`. Once a thread calls the `wait` method, it enters a *wait set* for the object. The thread is not made runnable when the lock is available. Instead, it stays deactivated until another thread has called the `notifyAll` method on the same object.

When another thread has added an element, it should call that method:

Click here to view code image

```
public synchronized void add(Object newValue) {
...
notifyAll();
}
```

The call to `notifyAll` reactivates all threads in the wait set. When the threads are removed from the wait set, they are again runnable and the scheduler will eventually activate them again. At that time, they will attempt to reacquire the lock. As one of them succeeds, it continues where it left off, returning from the call to `wait`.

At this time, the thread should test the condition again. There is no guarantee that the condition is now fulfilled—the `notifyAll` method merely signals to the waiting threads that it *may be* fulfilled at this time and that it is worth checking for the condition again. For that reason, the test is in a loop

```
while (head == null) wait();
```

A thread can only call `wait`, `notifyAll`, or `notify` on an object if it holds the lock on that object.

Caution

Another method, `notify`, unblocks only a single thread from the wait set. That is more efficient than unblocking all threads, but there is a danger. If the chosen thread finds that it still cannot proceed, it becomes blocked again. If no other thread calls `notify` again, the program deadlocks.

Note

> When implementing data structures with blocking methods, the `wait`, `notify`, and `notifyAll` methods are appropriate. But they are not easy to use properly. Application programmers should never have a need to use these methods. Instead, use prebuilt data structures such as `LinkedBlockingQueue` or `ConcurrentHashMap`.

10.8 Threads

As we are nearing the end of this chapter, the time has finally come to talk about threads, the primitives that actually execute tasks. Normally, you are better off using executors that manage threads for you, but the following sections give you some background information about working directly with threads.

10.8.1 Starting a Thread

Here is how to run a thread in Java:

Click here to view code image

```
Runnable task = () -> { ... };
Thread thread = new Thread(task);
thread.start();
```

The static `sleep` method makes the current thread sleep for a given period, so that some other threads have a chance to do work.

```
Runnable task = () -> {
...
Thread.sleep(millis);
...
}
```

If you want to wait for a thread to finish, call the `join` method:

```
thread.join(millis);
```

These two methods throw the checked `InterruptedException` that is discussed in the next section.

A thread ends when its `run` method returns, either normally or because an exception was thrown. In the latter case, the *uncaught exception handler* of the thread is invoked. When the thread is created, that handler is set to the uncaught exception handler of the thread group, which is ultimately the global handler (see Chapter 5). You can change the handler of a thread by calling the `setUncaughtExceptionHandler` method.

Note

The initial release of Java defined a `stop` method that immediately terminates a thread, and a `suspend` method that blocks a thread until another thread calls `resume`. Both methods have since been deprecated.

The `stop` method is inherently unsafe. Suppose a thread is stopped in the middle of a critical section—for example, inserting an element into a queue. Then the queue is left in a partially updated state. However, the lock protecting the critical section is unlocked, and other threads can use the corrupted data structure. You should interrupt a thread when you want it to stop. The interrupted thread can then stop when it is safe to do so.

The `suspend` method is not as risky but still problematic. If a thread is suspended while it holds a lock, any other thread trying to acquire that lock blocks. If the resuming thread is among them, the program deadlocks.

10.8.2 Thread Interruption

Suppose that, for a given query, you are always satisfied with the first result. When the search for an answer is distributed over multiple tasks, you want to cancel all others as soon as the answer is obtained. In Java, task cancellation is *cooperative*.

Each thread has an *interrupted status* that indicates that someone would like to “interrupt” the thread. There is no precise definition of what interruption means, but most programmers use it to indicate a cancellation request.

A `Runnable` can check for this status, which is typically done in a loop:

Click here to view code image

```
Runnable task = () -> {
while (more work to do) {
if (Thread.currentThread().isInterrupted()) return;
Do more work
}
};
```

When the thread is interrupted, the `run` method simply ends.

Note

There is also a static `Thread.interrupted` method which gets the interrupted status of the current thread, then clears it, and returns the old status.

Sometimes, a thread becomes temporarily inactive. That can happen if a thread waits for a value to be computed by another thread or for input/output, or if it goes to sleep to give other threads a chance.

If the thread is interrupted while it waits or sleeps, it is immediately reactivated —but in this case, the interrupted status is not set. Instead, an `InterruptedException` is thrown. This is a checked exception, and you must catch it inside the `run` method of a `Runnable`. The usual reaction to the exception is to end the `run` method:

Click here to view code image

```
Runnable task = () -> {
try {
while (more work to do) {
Do more work
Thread.sleep(millis);
}
}
catch (InterruptedException ex) {
// Do nothing
}
};
```

When you catch the `InterruptedException` in this way, there is no need to check for the interrupted status. If the thread was interrupted outside the call to `Thread.sleep`, the status is set and the `Thread.sleep` method throws an `InterruptedException` as soon as it is called.

Tip

The `InterruptedException` may seem pesky, but you should not just catch and hide it when you call a method such as `sleep`. If you can't do anything else, at least set the interrupted status:

Click here to view code image

```
try {
Thread.sleep(millis);
} catch (InterruptedException ex) {
Thread.currentThread().interrupt();
}
```

Or better, simply propagate the exception to a competent handler:

Click here to view code image

```
public void mySubTask() throws InterruptedException {
...
Thread.sleep(millis);
...
}
```

10.8.3 Thread-Local Variables

Sometimes, you can avoid sharing by giving each thread its own instance, using the `ThreadLocal` helper class. For example, the `NumberFormat` class is not threadsafe. Suppose we have a static variable

Click here to view code image

```
public static final NumberFormat currencyFormat =
NumberFormat.getCurrencyInstance();
```

If two threads execute an operation such as

Click here to view code image

```
String amountDue = currencyFormat.format(total);
```

then the result can be garbage since the internal data structures used by the `NumberFormat` instance can be corrupted by concurrent access. You could use a lock or provide a synchronized method to ensure atomic access to the shared `NumberFormat` variable. Alternatively, you could construct a local `NumberFormat` object whenever you need it, but that is also wasteful.

To construct one instance per thread, use the following code:

Click here to view code image

```
public static final ThreadLocal<NumberFormat> currencyFormat
= ThreadLocal.withInitial(() -> NumberFormat.getCurrencyInstance());
```

To access the actual formatter, call

Click here to view code image

```
String amountDue = currencyFormat.get().format(total);
```

The first time you call `get` in a given thread, the lambda expression in the constructor is called to create the instance for the thread. From then on, the `get` method returns the instance belonging to the current thread.

10.8.4 Miscellaneous Thread Properties

The `Thread` class exposes a number of properties for threads, but most of them are more useful for students of certification exams than application programmers. This section briefly reviews them.

Threads can be collected in groups, and there are API methods to manage thread groups, such as interrupting all threads in a group. Nowadays, executors are the preferred mechanism for managing groups of tasks.

You can set *priorities* for threads, where high-priority threads are scheduled to run before lower-priority ones. Hopefully, priorities are honored by the virtual machine and the host platform, but the details are highly platform-dependent. Therefore, using priorities is fragile and not generally recommended.

Threads have *states*, and you can tell whether a thread is new, running, blocked on input/output, waiting, or terminated. When you use threads as an application programmer, you rarely have a reason to inquire about their states.

Threads have names, and you can change the name for debugging purposes. For example:

Click here to view code image

```
Thread.currentThread().setName("Bitcoin-miner-1");
```

When a thread terminates due to an uncaught exception, the exception is passed to the thread's *uncaught exception handler*. By default, its stack trace is dumped to `System.err`, but you can install your own handler (see Chapter 5).

A *daemon* is a thread that has no other role in life than to serve others. This is useful for threads that send timer ticks or clean up stale cache entries. When only daemon threads remain, the virtual machine exits.

To make a daemon thread, call `thread.setDaemon(true)` before starting the thread.

10.9 Processes

Up to now, you have seen how to execute Java code in separate threads within the same program. Sometimes, you need to execute another program. For this, use the `ProcessBuilder` and `Process` classes. The `Process` class executes a command in a separate operating system process and lets you interact with its standard input, output, and error streams. The `ProcessBuilder` class lets you configure a `Process` object.

Note

The ProcessBuilder class is a more flexible replacement for the Runtime.exec calls.

10.9.1 Building a Process

Start the building process by specifying the command that you want to execute. You can supply a List<String> or simply the strings that make up the command.

Click here to view code image

```
ProcessBuilder builder = new ProcessBuilder("gcc", "myapp.c");
```

Caution

The first string must be an executable command, not a shell builtin. For example, to run the dir command in Windows, you need to build a process with strings "cmd.exe", "/C", and "dir".

Each process has a *working directory*, which is used to resolve relative directory names. By default, a process has the same working directory as the virtual machine, which is typically the directory from which you launched the java program. You can change it with the directory method:

Click here to view code image

```
builder = builder.directory(path.toFile());
```

Note

Each of the methods for configuring a ProcessBuilder returns itself, so that you can chain commands. Ultimately, you will call

Click here to view code image

```
Process p = new ProcessBuilder(command).directory(file).start();
```

Next, you will want to specify what should happen to the standard input, output, and error streams of the process. By default, each of them is a pipe that you can access with

Click here to view code image

```
OutputStream processIn = p.getOutputStream();
InputStream processOut = p.getInputStream();
InputStream processErr = p.getErrorStream();
```

Note that the input stream of the process is an output stream in the JVM! You write to that stream, and whatever you write becomes the input of the process. Conversely, you read what the process writes to the output and error streams. For you, they are input streams.

You can specify that the input, output, and error streams of the new process should be the same as the JVM. If the user runs the JVM in a console, any user input is forwarded to the process, and the process output shows up in the console. Call

```
builder.inheritIO()
```

to make this setting for all three streams. If you only want to inherit some of the streams, pass the value

Click here to view code image

```
ProcessBuilder.Redirect.INHERIT
```

to the `redirectInput`, `redirectOutput`, or `redirectError` methods. For example,

Click here to view code image

```
builder.redirectOutput(ProcessBuilder.Redirect.INHERIT);
```

You can redirect the process streams to files by supplying `File` objects:

Click here to view code image

```
builder.redirectInput(inputFile)
.redirectOutput(outputFile)
.redirectError(errorFile)
```

The files for output and error are created or truncated when the process starts. To append to existing files, use

Click here to view code image

```
builder.redirectOutput(ProcessBuilder.Redirect.appendTo(outputFile));
```

It is often useful to merge the output and error streams, so you see the outputs and error messages in the sequence in which the process generates them. Call

Click here to view code image

```
builder.redirectErrorStream(true)
```

to activate the merging. If you do that, you can no longer call `redirectError` on the `ProcessBuilder` or `getErrorStream` on the `Process`.

Finally, you may want to modify the environment variables of the process. Here, the builder chain syntax breaks down. You need to get the builder's environment (which is initialized by the environment variables of the process running the JVM), then put or remove entries.

Click here to view code image

```
Map<String, String> env = builder.environment();
env.put("LANG", "fr_FR");
env.remove("JAVA_HOME");
Process p = builder.start();
```

10.9.2 Running a Process

After you have configured the builder, invoke its `start` method to start the process. If you configured the input, output, and error streams as pipes, you can now write to the input stream and read the output and error streams. For example,

Click here to view code image

```
Process process = new ProcessBuilder("/bin/ls", "-l")
.directory(Paths.get("/tmp").toFile())
.start();
try (Scanner in = new Scanner(process.getInputStream())) {
while (in.hasNextLine())
System.out.println(in.nextLine());
}
```

Caution

There is limited buffer space for the process streams. You should not flood the input, and you should read the output promptly. If there is a lot of input and output, you may need to produce and consume it in separate threads.

To wait for the process to finish, call

Click here to view code image

```
int result = process.waitFor();
```

or, if you don't want to wait indefinitely,

Click here to view code image

```
long delay = ...;
if (process.waitfor(delay, TimeUnit.SECONDS)) {
int result = process.exitValue();
...
} else {
process.destroyForcibly();
}
```

The first call to `waitFor` returns the exit value of the process (by convention, `0` for success or a nonzero error code). The second call returns `true` if the process didn't time out. Then you need to retrieve the exit value by calling the `exitValue` method.

Instead of waiting for the process to finish, you can just leave it running and occasionally call `isAlive` to see whether it is still alive. To kill the process, call `destroy` or `destroyForcibly`. The difference between these calls is platform-dependent. On Unix, the former terminates the process with `SIGTERM`, the latter with `SIGKILL`. (The `supportsNormalTermination` method returns `true` if the `destroy` method can terminate the process normally.)

Finally, you can receive an asynchronous notification when the process has completed. The call `process.onExit()` yields a `CompletableFuture<Process>` that you can use to schedule any action.

Click here to view code image

```
process.onExit().thenAccept(
p -> System.out.println("Exit value: " + p.exitValue()));
```

10.9.3 Process Handles

To get more information about a process that your program started, or any other process that is currently running on your machine, use the `ProcessHandle` interface. You can obtain a `ProcessHandle` in four ways:

1. Given a `Process` object `p`, `p.toHandle()` yields its `ProcessHandle`.
2. Given a `long` operating system process ID, `ProcessHandle.of(id)` yields the handle of that process.
3. `ProcessHandle.current()` is the handle of the process that runs this Java virtual machine.
4. `ProcessHandle.allProcesses()` yields a `Stream<ProcessHandle>` of all operating system processes that are

visible to the current process.

Given a process handle, you can get its process ID, its parent process, its children, and its descendants:

Click here to view code image

```
long pid = handle.pid();
Optional<ProcessHandle> parent = handle.parent();
Stream<ProcessHandle> children = handle.children();
Stream<ProcessHandle> descendants = handle.descendants();
```

Note

The `Stream<ProcessHandle>` instances that are returned by the `allProcesses`, `children`, and `descendants` methods are just snapshots in time. Any of the processes in the stream may be terminated by the time you get around to seeing them, and other processes may have started that are not in the stream.

The `info` method yields a `ProcessHandle.Info` object with methods for obtaining information about the process.

Click here to view code image

```
Optional<String[]> arguments()
Optional<String> command()
Optional<String> commandLine()
Optional<String> startInstant()
Optional<String> totalCpuDuration()
Optional<String> user()
```

All of these methods return `Optional` values since it is possible that a particular operating system may not be able to report the information.

For monitoring or forcing process termination, the `ProcessHandle` interface has the same `isAlive`, `supportsNormalTermination`, `destroy`, `destroyForcibly`, and `onExit` methods as the `Process` class. However, there is no equivalent to the `waitFor` method.

Exercises

1. Using parallel streams, find all files in a directory that contain a given word. How do you find just the first one? Are the files actually searched concurrently?