Chapter 13
Concurrency

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- ✓ **Managing Concurrent Code Execution**
  - Create both platform and virtual threads. Use both Runnable and Callable objects, manage the thread lifecycle, and use different Executor services and concurrent API to run tasks.
  - Develop thread-safe code, using locking mechanisms and concurrent API.
  - Process Java collections concurrently and utilize parallel streams.
- ✓ **Working with Streams and Lambda expressions**
  - Perform decomposition, concatenation, and reduction, and grouping and partitioning on sequential and parallel streams.

As you will learn in Chapter 14, "I/O," computers are capable of reading and writing data. Unfortunately, these disk/network operations are much slower than CPU operations. In fact, if your computer's operating system were to stop and wait for every disk or network operation to finish, your computer would appear to freeze constantly.

Luckily, all operating systems support what is known as *multithreaded processing* where an application or group of applications can execute multiple tasks at the same time. This allows tasks waiting to give way to other processing requests.

In this chapter, we introduce threads and provide numerous ways to manage threads using the Concurrency API. Threads and concurrency are challenging topics for many programmers to grasp, as problems with threads can be frustrating even for veteran developers. In practice,

concurrency issues are among the most difficult problems to diagnose and resolve.

Introducing Threads

We begin this chapter by reviewing common terminology associated with threads. A *thread* is the smallest unit of execution that can be scheduled by the operating system. A *process* is a group of associated threads that execute in the same shared environment. It follows, then, that a *single-threaded process* is one that contains exactly one thread, whereas a *multithreaded process* contains one or more threads.

By *shared environment*, we mean that the threads in the same process share the same memory space and can communicate directly with one another.

Within a computer, an Operating System (OS) manages the operating system threads using the underlying CPU hardware. Java executes processes using *platform threads*, which have a one-to-one mapping with operating system threads, as shown in Figure 13.1.

This figure shows a single process with three platform threads. It also shows how they are mapped to an arbitrary number of *n* CPUs available within the system.

The Java Virtual Machine (JVM) creates and manages two different types of platform threads. A *system thread* is created by the JVM and runs in the background of the application. For example, garbage collection is managed by a system thread created by the JVM. Alternatively, a *user-defined thread* is created by the application developer.

In this chapter, we talk a lot about tasks and their relationships to threads. A *task* is a single unit of work performed by a thread. A thread can complete multiple independent tasks but only one task at a time.

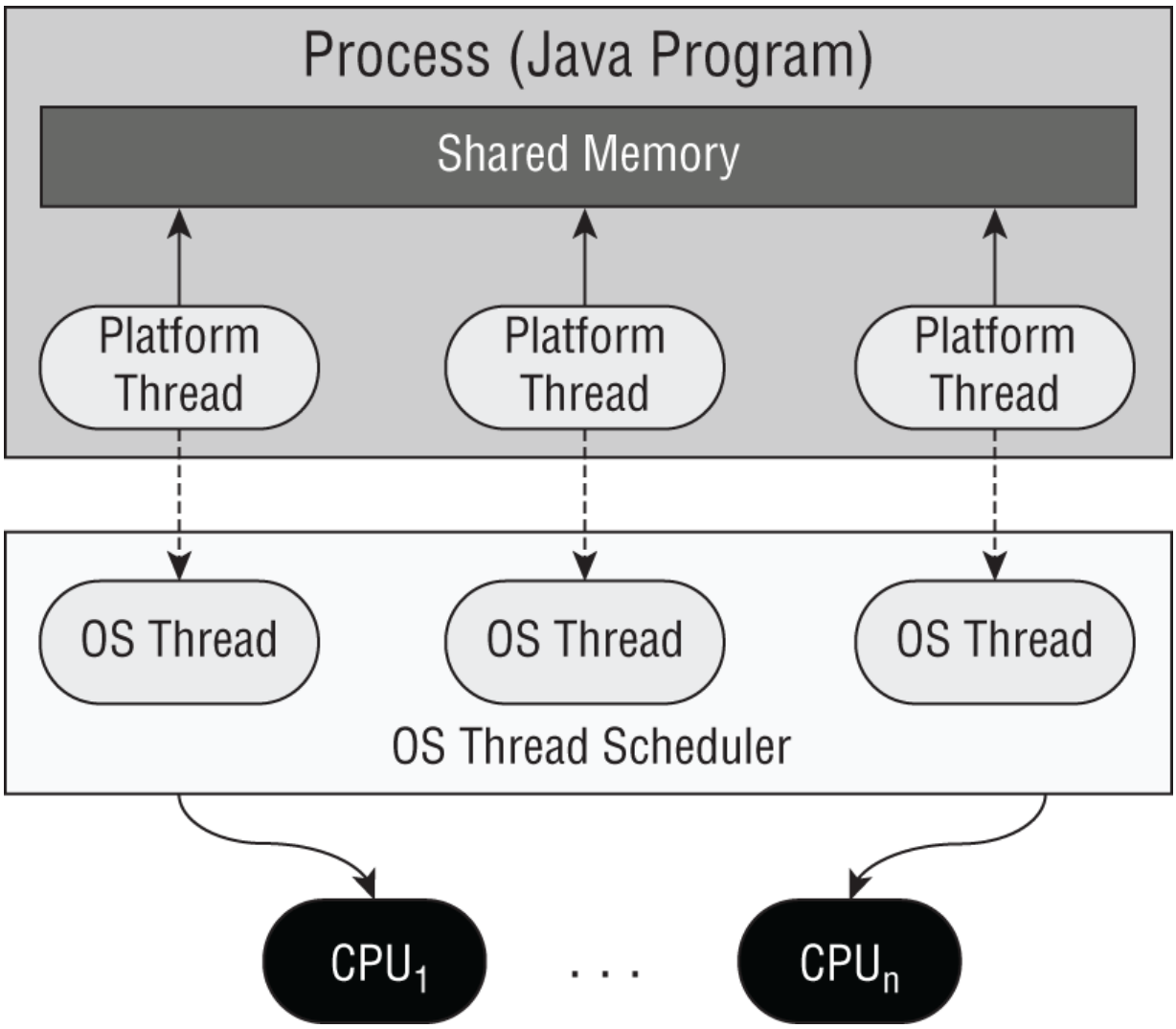

FIGURE 13.1 Platform threads

By *shared memory* in Figure 13.1, we are generally referring to `static` variables as well as instance and local variables passed to a thread. Yes, you finally see how `static` variables can be useful for performing complex, multithreaded tasks! Remember from Chapter 5, "Methods," that `static` methods and variables are defined on a single class object. For example, if one thread updates the value of a `static` variable, this information is immediately available for other threads within the process to read.

Comparing to Virtual Threads

Platform threads are often inefficient. They are like having a personal butler who stands around in case you need something. If you constantly need

things, this is a good use of the butler's time. For a platform thread to be efficient, you need to be heavily using the CPU.

By contrast, when we go to a restaurant, there is a server who is assigned to many tables. Since we don't need someone to stand there while the food is cooking and when we eat, this is a more efficient use of the server's time. The Java equivalent of a single server handling multiple tables is a *carrier thread*. The tables correspond to *virtual threads*, which are less resource intensive than platform threads, making virtual threads a good choice when you expect to wait for I/O or network resources.

Each time the virtual thread is ready to run, it waits for a carrier thread to be available. The virtual thread does not automatically get the same carrier thread. It's like when a server walks by and refills your coffee or water. They don't tell you that only your original server can do it! Figure 13.2 shows a virtual thread running on a carrier thread.

Notice how the other carrier thread is not currently running a virtual thread. The three blocked virtual threads are not ready to continue, so they don't need a carrier thread at the moment. They are not tied to specific OS threads, which frees the OS threads to work on other tasks. (Thanks to Venkat Subramaniam for the waiter analogy. The butler extension is all ours!)

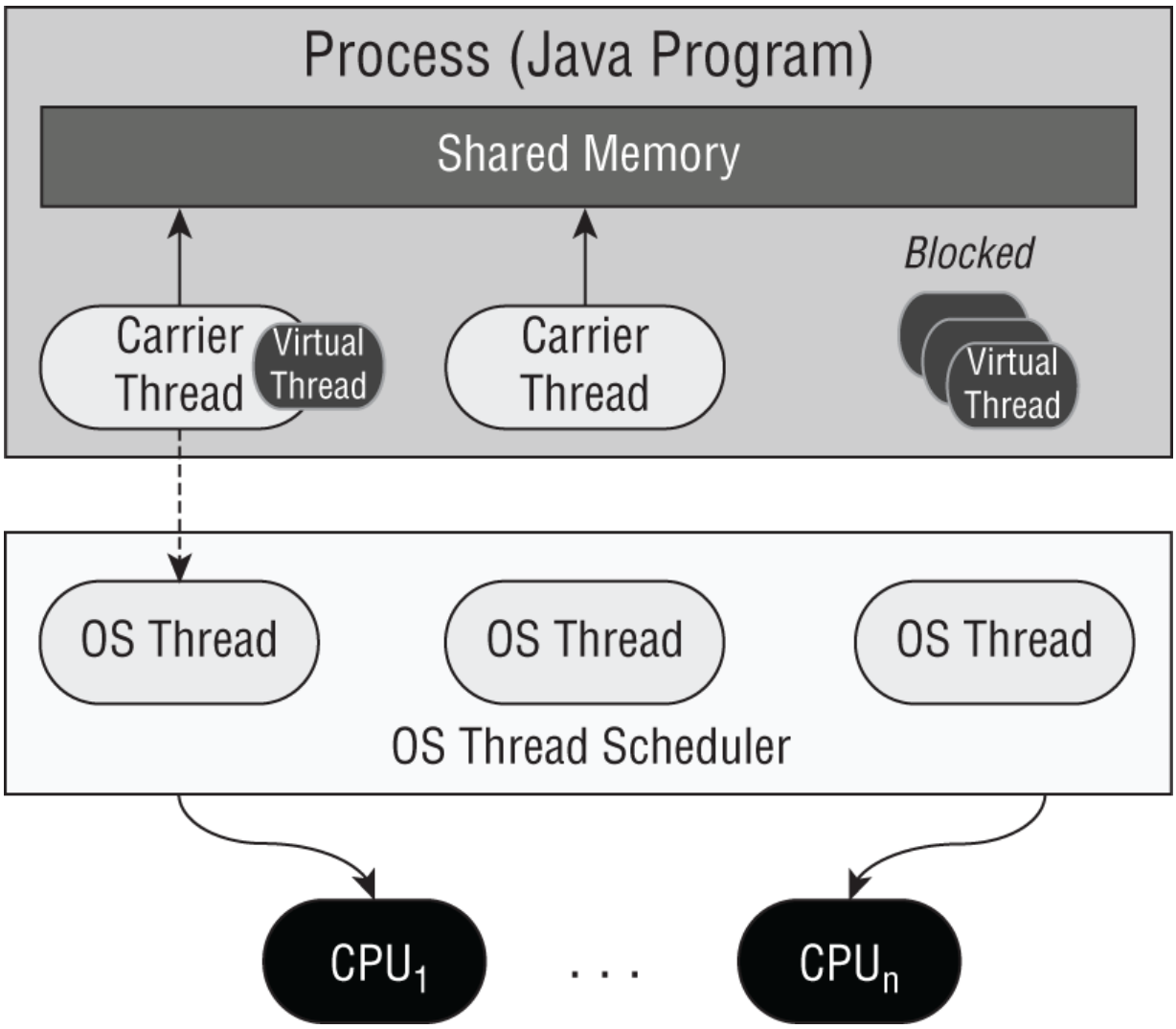

FIGURE 13.2 Virtual threads

In documentation, you may see a carrier thread referenced as a type of platform thread. This is intending to convey that it runs on the operating system. From a code point of view, platform threads and carrier threads are different things.

Real World Scenario

Platform vs. Virtual Threads

To see how lightweight virtual threads are, let's compare them to platform threads. Don't worry if you haven't seen these methods before. You'll learn about them later in this chapter. Suppose we have a task that takes a second to run:

```
public class PlatformVsVirtual {
   static void waitUp() {
      try {
         Thread.sleep(1_000);
      } catch (InterruptedException e) {
         throw new RuntimeException(e);
      }
   } }
```

Now we try to run a million platform threads. How long do you think it takes?

```
   public static void main(String[] args) throws
InterruptedException {
      var threads = Stream.generate(() ->
Thread.ofPlatform()
         .unstarted(PlatformVsVirtual::waitUp))
         .limit(1_000_000)
         .toList();
      threads.forEach(Thread::start);
      for (var t : threads)
         t.join();
   }
```

The answer is that it may not even run. Platform threads are resource intensive enough that the program likely fails.

```
Exception in thread "main" java.lang.OutOfMemoryError:
unable to create native thread: possibly out of memory
or process/resource limits reached
```

Changing `Thread.ofPlatform()` to `Thread.ofVirtual()` allows the program to run. The program was pretty fast when we tested it. That's pretty good for a million threads!

Understanding Thread Concurrency

The property of executing multiple threads and processes at the same time is referred to as *concurrency*. A *thread scheduler* determines which threads should be currently executing. For example, a thread scheduler may employ a *round-robin schedule* in which each available thread receives an equal number of CPU cycles with which to execute, with threads visited in a circular order.

When a thread's allotted time is complete but the thread has not finished processing, a context switch occurs. A *context switch* is the process of storing a thread's current state and later restoring the state of the thread to continue execution. Since there's a cost to context switch due to lost time and having to reload a thread's state, intelligent thread schedulers do their best to minimize the number of context switches while keeping an application running smoothly.

Finally, a thread can interrupt or supersede another thread if it has a higher thread priority. A *thread priority* is a numeric value associated with a thread that the thread scheduler considers when determining which threads should execute. The priority can be set from 1 (`Thread.MIN_PRIORITY`) to 10 (`Thread.MAX_PRIORITY`), either before a thread is started or while it is running.

```
var thread1 = new Thread(() -> System.out.print("Super
Important"));
thread1.setPriority(Thread.MAX_PRIORITY);
thread1.start();

var thread2 = new Thread(() -> System.out.print("Less
Important"));
thread2.start();
thread2.setPriority(2);
```

Note that thread priority is just a suggestion and the JVM does not guarantee the order that the threads will execute. For example, if there are

enough thread resources available for all processes, then all threads will run at the same time.

Creating a Thread

One of the most common ways to define a task for a thread is by using the `Runnable` interface, which takes no arguments and returns no data.

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

With this, it's easy to create and start a thread. In fact, you can do so in one line of code using the `Thread` class:

```
Thread.ofPlatform().start(() -> System.out.print("Hello"));
System.out.print("World");
```

The first line creates a new platform thread builder object with the `ofPlatform()` method. It then starts the thread with the `start()` method, while also passing in a `Runnable` task, aka the work to be done, as a lambda expression. Does this code print `HelloWorld` or `WorldHello`? The answer is that we don't know. Depending on the thread priority/scheduler, either is possible. Remember that order of thread execution is not often guaranteed. The exam commonly presents questions in which multiple tasks are started at the same time, and you must determine the result.

There are a number of ways to create a thread, as shown in Table 13.1. Pay attention to which are platform threads and which are virtual threads.

TABLE 13.1 Creating and starting a `Thread`

Code	Type	Description
`var builder = Thread.ofPlatform();` `Thread thread = builder.start(runnable);`	Platform	Factory
`var builder = Thread.ofVirtual();` `Thread thread = builder.start(runnable);`	Virtual	Factory
`Thread thread = new Thread(runnable);` `thread.start();`	Platform	Constructor

Prior to Java 21, it was advisable to create a thread with the `new Thread()` constructor call, in which the task is defined when the thread is created. Starting with Java 21, the factory method is preferable since it is clearer which type of thread you are getting. The factory method creates a builder, which allows you to call other methods to set attributes like the name.

For platform threads, you can set a priority via the builder object by calling `priority()`. The priority for virtual threads is always 5 (`Thread.NORM_PRIORITY`) and cannot be changed. Calling `setPriority()` on the newly created virtual `Thread` has no effect.

Deferring the Task

Notice that `Thread.ofPlatform()` and `Thread.ofVirtual()` create a builder that you use to create a `Thread`. You pass a `Runnable` using the `start()` or `unstarted()` method depending on whether you want the thread to run now or later. Take a look at the following code snippet, which starts the thread after the task has been set:

```
Thread t = Thread.ofVirtual().unstarted(task);

// Do some other stuff

t.start();
```

Let's take a look at a more complex example. What is the output of this?

```
12: public static void main(String[] args)
13:       throws InterruptedException {
14:    Runnable printInventory =
15:       () -> System.out.println("Printing zoo inventory");
16:    Runnable printRecords = () -> {
17:       for (int i = 0; i < 3; i++)
18:          System.out.println("Printing record: " + i);
19:    };
20:    System.out.println("begin");
21:    var platformThread = Thread.ofPlatform()
22:       .priority(10)
23:       .start(printInventory);
24:    var virtualThread = Thread.ofVirtual()
```

```
25:        .start(printRecords);
26:    var constructorThread = new Thread(printInventory);
27:    constructorThread.start();
28:    System.out.println("end");
29:    platformThread.join();
30:    virtualThread.join();
31:    constructorThread.join();
32: }
```

The answer is that the order is unknown until runtime. The following is just one possible output:

```
begin
Printing record: 0
Printing zoo inventory
end
Printing record: 1
Printing zoo inventory
Printing record: 2
```

This sample uses a total of four threads: the `main()` user thread and three additional threads created on lines 21–25. Each thread created on these lines is executed as an asynchronous task. By *asynchronous*, we mean that the thread executing the `main()` method does not wait for the results of each newly created thread before continuing. The opposite of this behavior is a *synchronous* task in which the program waits (or *blocks*) on line 20 for the thread to finish executing before moving on to the next line. So far, the vast majority of method calls used in this book have been synchronous. Remember that the `priority()` call is a suggestion, so you cannot assume it influences the order of the threads.

While the order of thread execution is indeterminate once the threads have been started, the order within a single thread is still linear. In particular, the `for()` loop is still ordered. Also, `begin` always appears before `end`.

The `join()` methods on lines 29–31 tell the `main()` method not to end before the three threads have completed. The `join()` method throws an `InterruptedException` if it fails, which the `main()` method declares.

Calling *run()* Instead of *start()*

On the exam, be mindful of code that attempts to start a thread by calling `run()` instead of `start()`. Calling `run()` on a `Thread` or a `Runnable` *does not start a new thread*. While the following code snippet will compile, it runs synchronously rather than starting a thread:

```
new Thread(printInventory).run();
```

Working with Daemon Threads

A *daemon thread* is one that will not prevent the JVM from exiting when the program finishes. A Java application terminates when the only threads that are running are daemon threads. For example, if garbage collection is the only thread left running, the JVM will automatically shut down.

Let's take a look at an example. What do you think this outputs?

```
1:  public class Zoo {
2:     public static void pause() {                 // Defines
the thread task
3:        try {
4:           Thread.sleep(10_000);                  // Wait for
10 seconds
5:        } catch (InterruptedException e) {}
6:        System.out.println("Thread finished!");
7:     }
8:
9:     public static void main(String[] unused) {
10:       var job = Thread.ofPlatform().start(Zoo::pause);
11:       System.out.println("Main method finished!");
12:    } }
```

The program will output two statements roughly 10 seconds apart:

```
Main method finished!
< 10 second wait >
Thread finished!
```

That's right. Even though the `main()` method is done, the JVM will wait for the user thread to be done before ending the program. What if we change

`job` to be a daemon thread by adding this to line 11?

```
10: var job =
Thread.ofPlatform().daemon(true).start(Zoo::pause);
```

The program will print the first statement and terminate without ever printing the second line.

```
Main method finished!
```

Virtual threads are always daemons. Platform threads default to non-daemon but can be changed.

Managing a Thread's Life Cycle

After a `Thread` object has been created, it is in one of six states, shown in Figure 13.3. You can query a thread's state by calling `getState()` on the thread object.

Every thread is initialized with a `NEW` state. As soon as `start()` is called, the thread is moved to a `RUNNABLE` state. Does that mean it is actually running? Not exactly: it may be running, or it may not be. The `RUNNABLE` state just means the thread is able to be run. Once the work for the thread is completed or an uncaught exception is thrown, the thread state becomes `TERMINATED`, and no more work is performed.

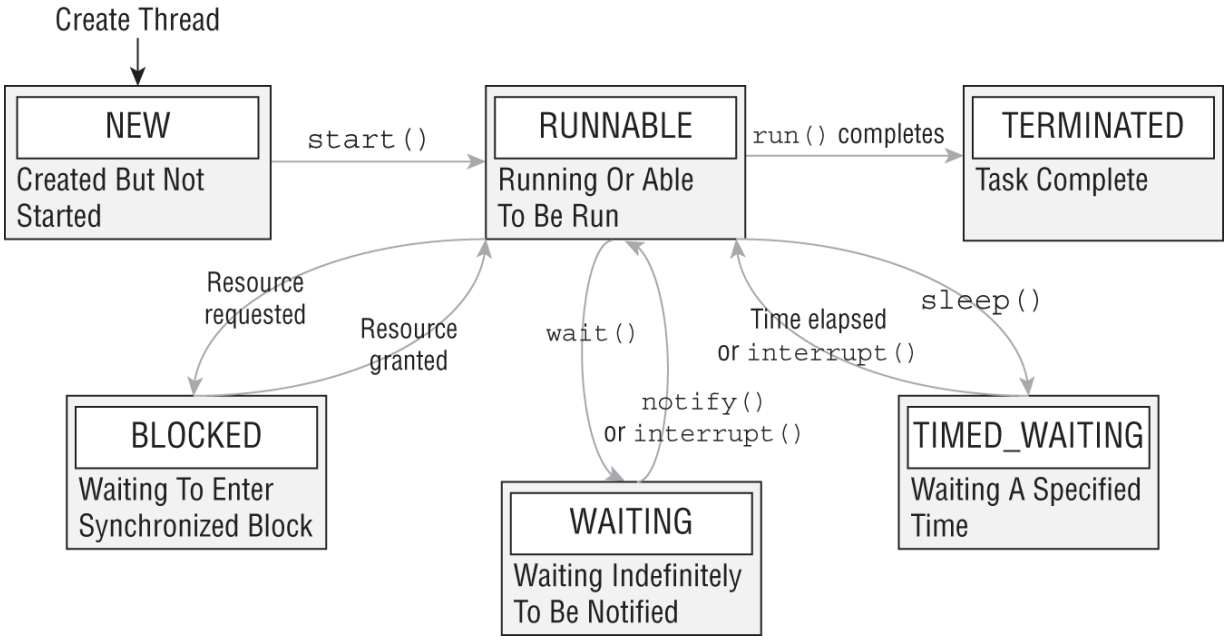

FIGURE 13.3 Thread states

While in a `RUNNABLE` state, the thread may transition to one of three states where it pauses its work: `BLOCKED`, `WAITING`, or `TIMED_WAITING`. In these states, a thread is not using any CPU resources (other than keeping track of a timer for `TIMED_WAITING`). Figure 13.3 includes common transitions between thread states, but there are other possibilities. For example, a thread in a `WAITING` state might be triggered by `notifyAll()`.

Once a thread enters a waiting state, another thread can call `interrupt()` on the thread, causing it to move back to a `RUNNABLE` state. Most methods that cause a thread to wait also declare `InterruptedException`.

```
var thread = Thread.ofPlatform().start(() -> {
   try {
      Thread.sleep(1000);
   } catch (InterruptedException e) {
      System.out.println("Interrupted!");
   }});
thread.interrupt();
```

This code prints `Interrupted!` and then resumes running. What if `interrupt()` is called on a thread that is already in the `RUNNABLE` state? In this case, an exception won't be thrown. The thread can periodically check `Thread.interrupted()` to determine if an interrupt has been sent recently.

```
var thread = Thread.ofPlatform().start(() -> {
   while(true) {
      if(Thread.interrupted())
         System.out.println("Someone interrupted us!");
      }});
thread.interrupt();
```

We cover some (but not all) of the transitions in Figure 13.3 in this chapter. If an operation that is not supported is called on a thread, such as interrupting a suspended thread, an `IllegalThreadStateException` will be thrown. Some thread-related methods—such as `wait()`and `notify()`—are beyond the scope of the exam and, frankly, difficult to use well. You should avoid them and use the Concurrency API as much as possible. It takes a large amount of skill (and some luck!) to use these methods correctly.

Reviewing Thread Concepts

We conclude this section with the list of terminology that you should know in Table 13.2. Pay close attention to the different types of threads, as we'll be using them throughout the chapter.

TABLE 13.2 Java thread terminology

Term	Description
Carrier thread	System thread that runs virtual threads when they are not blocked
Daemon thread	Thread that will not prevent the JVM from exiting when the program finishes
Platform thread	Thread that is scheduled by the operating system
Process	Group of associated threads that execute in the same shared environment
System thread	Thread created by the JVM that runs in the background of the application, such as the garbage collector
Task	Single unit of work performed by a thread
Thread	Smallest unit of execution that can be scheduled
User-defined thread	Thread created by the application developer to accomplish a specific task
Virtual thread	Lightweight thread that is mapped to a carrier thread when needed to run

Creating Threads with the Concurrency API

Java includes the `java.util.concurrent` package, which we refer to as the Concurrency API, to handle the complicated work of managing threads for you. The Concurrency API includes the `ExecutorService` interface, which defines services that create and manage threads.

You first obtain an instance of an `ExecutorService` interface, and then you send the service tasks to be processed. The framework includes numerous useful features, such as thread pooling and scheduling. It is recommended that you use this framework any time you need to create and execute a separate task, even if you need only a single thread.

Introducing the Single-Thread Executor

Since `ExecutorService` is an interface, how do you obtain an instance of it? The Concurrency API includes the `Executors` factory class that can be used to create instances of the `ExecutorService` object. Let's rewrite our earlier example with the two `Runnable` instances to using an `ExecutorService`.

```
try (ExecutorService service =
Executors.newSingleThreadExecutor()) {
   System.out.println("begin");
   service.execute(printInventory);
   service.execute(printRecords);
   service.execute(printInventory);
   System.out.println("end");
}
```

In this example, we use the `newSingleThreadExecutor()` method to create the service. Unlike our earlier example, in which we had four threads (one `main()` and three new threads), we have only two threads (one `main()` and one new thread). This means that the output, while still unpredictable, will have less variation than before. For example, the following is one possible output:

```
begin
Printing zoo inventory
Printing record: 0
Printing record: 1
end
Printing record: 2
Printing zoo inventory
```

Notice that the `printRecords` loop is no longer interrupted by other `Runnable` tasks sent to the thread executor. With a single-thread executor, tasks are guaranteed to be executed sequentially. Notice that the `end` text is output while our thread executor tasks are still running. This is because the `main()` method is still an independent thread from the `ExecutorService`.

Submitting Tasks

You can submit tasks to an `ExecutorService` instance multiple ways. The `execute()` method takes a `Runnable` instance and completes the task asynchronously. Because the return type of the method is `void`, it does not tell us anything about the result of the task. It is considered a "fire-and-

forget” method, as once it is submitted, the results are not directly available to the calling thread.

Fortunately, the writers of Java added `submit()` methods to the `ExecutorService` interface, which, like `execute()`, can be used to complete tasks asynchronously. Unlike `execute()`, though, `submit()` returns a `Future` instance that can be used to determine whether the task is complete. It can also be used to return a generic result object after the task has been completed.

Table 13.3 shows the five methods, including `execute()` and two `submit()` methods, that you should know for the exam. Don’t worry if you haven’t seen `Future` or `Callable` before; we discuss them in detail in the next section.

TABLE 13.3 `ExecutorService` methods

Method name	Description
`void` **`execute`**`(Runnable command)`	Executes `Runnable` task at some point in future.
`Future<?>` **`submit`**`(Runnable task)`	Executes `Runnable` task at some point in future and returns `Future` representing task.
`<T> Future<T>` **`submit`**`(Callable<T> task)`	Executes `Callable` task at some point in future and returns `Future` representing pending results of task.
`<T> List<Future<T>>` **`invokeAll`**`(Collection<? extends Callable<T>> tasks)`	Executes given tasks and waits for all tasks to complete. Returns `List` of `Future` instances in the same order in which they were in original collection.
`<T> T` **`invokeAny`**`(Collection<? extends Callable<T>> tasks)`	Executes given tasks and waits for at least one to complete.

For the exam, you need to be familiar with both `execute()` and `submit()`, but in your own code we recommend `submit()` over `execute()` whenever possible since it supports `Callable` and therefore an optional return value.

Waiting for Results

How do we know when a task submitted to an `ExecutorService` is complete? As mentioned in the previous section, the `submit()` method returns a `Future<?>` instance that can be used to determine this result.

```
Future<?> future = service.submit(() ->
System.out.println("Hello"));
```

The `Future` type is actually an interface. For the exam, you don't need to know any of the classes that implement `Future`, just that a `Future` instance is returned by various API methods. Table 13.4 includes the most useful methods.

TABLE 13.4 `Future` methods

Method name	Description
`boolean isDone()`	Returns `true` if task was completed, threw exception, or was cancelled.
`boolean isCancelled()`	Returns `true` if task was cancelled before it completed normally.
`boolean cancel(boolean mayInterruptIfRunning)`	Attempts to cancel execution of task and returns `true` if it was successfully cancelled or `false` if it could not be cancelled or is complete.
`V get()`	Retrieves result of task, waiting endlessly if it is not yet available.
`V get(long timeout, TimeUnit unit)`	Retrieves result of task, waiting specified amount of time. If result is not ready by time timeout is reached, checked `TimeoutException` will be thrown.

The following uses a `Future` instance to wait for the results:

```
import java.util.concurrent.*;
public class CheckResults {
   private static int counter = 0;
   public static void main(String[] unused) throws Exception {
      try (var service = Executors.newSingleThreadExecutor()) {
         Future<?> result = service.submit(() -> {
```

```
            for (int i = 0; i < 1_000_000; i++) counter++;
         });
         result.get(10, TimeUnit.SECONDS);  // Returns null for
Runnable
         System.out.println("Reached!");
      } catch (TimeoutException e) {
         System.out.println("Not reached in time");
      } } }
```

Note this example does not use the `Thread` class directly. In part, this is the essence of the Concurrency API: to do complex things with threads without having to manage threads directly. This code also waits at most 10 seconds, throwing a `TimeoutException` on the call to `result.get()` if the task is not done.

What is the return value of this task? As `Future<V>` is a generic interface, the type `V` is determined by the return type of the `Runnable` method. Since the return type of `Runnable.run()` is `void`, the `get()` method always returns `null` when working with `Runnable` expressions.

The `Future.get()` method can take an optional value and enum type `java.util.concurrent.TimeUnit`. Table 13.5 lists the full list of `TimeUnit` values since numerous methods in the Concurrency API use this enum.

TABLE 13.5 `TimeUnit` values

Enum name	Description
`TimeUnit.NANOSECONDS`	Time in one-billionths of a second (1/1,000,000,000)
`TimeUnit.MICROSECONDS`	Time in one-millionths of a second (1/1,000,000)
`TimeUnit.MILLISECONDS`	Time in one-thousandths of a second (1/1,000)
`TimeUnit.SECONDS`	Time in seconds
`TimeUnit.MINUTES`	Time in minutes
`TimeUnit.HOURS`	Time in hours
`TimeUnit.DAYS`	Time in days

Polling with Sleep

Polling is the process of intermittently checking data at some fixed interval. You might see code using `Thread.sleep()` inside of a loop for this purpose in older code. It is much better to use a `Future` and let Java handle the checking for you!

Investigating Callable

The `java.util.concurrent.Callable` functional interface is similar to `Runnable` except that its `call()` method returns a value and can throw a checked exception. The following is the definition of the `Callable` interface:

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

When `Callable<V>` is passed to an `ExecutorService` via `submit()`, a `Future<V>` object is returned. Once the task is complete, calling `get()` on the `Future<V>` will return the result of type `V`. This allows you to find out a lot of information about the results of the task.

The `ExecutorService submit()` method takes a `Runnable` and returns a `Future<?>` object. This object can be used to check if the thread is done. But, since `Runnable` tasks have a return type of `void`, calling `get()` on such a `Future` will always return `null` upon successful completion of the task.

The `Callable` interface is often preferable over `Runnable`, since it allows more details to be retrieved easily from the task after it is completed. That

said, we use both interfaces, as they are interchangeable in situations where the lambda does not throw an exception, and there is no return type. Luckily, the `ExecutorService` includes an overloaded version of the `submit()` method that takes a `Callable` object and returns a generic `Future<T>` instance.

Let's take a look at an example using `Callable`:

```
try (var service = Executors.newSingleThreadExecutor()) {
   Future<Integer> result = service.submit(() -> 30 + 11);
   System.out.println(result.get());   // 41
}
```

This implementation is easier to code and understand than if we had used a `Runnable`, some shared object, and an `interrupt()` or timed wait. In essence, that's the spirit of the Concurrency API, giving you the tools to write multithreaded code that is thread-safe, performant, and easy to follow.

Shutting Down a Thread Executor

You might have noticed the `ExecutorService` has been declared in a try-with-resources block like you learned about in Chapter 11, "Exceptions and Localization." A thread executor creates a *non-daemon* thread on the first task that is executed, so forgetting to do this will result in your application *never terminating*. Don't believe us? Try executing the following:

```
public class MissingClose {
   public static void main(String[] args) {
      var service = Executors.newSingleThreadExecutor();
      service.submit(() -> System.out.println("Never stops"));
   } }
```

This code runs but never terminates, because the `ExecutorService` is never shut down or closed. The fix is to always use an `ExecutorService` with a try-with-resources block. The try-with-resources block automatically calls `close()`, which shuts down the executor service so no more tasks get accepted. It then waits until they all complete execution.

Using an `ExecutorService` in a try-with-resources takes care of ensuring the tasks complete. However, the `close()` method was only added to this API in Java 19. In older code, you'd see an explicit `shutdown()` call. Or perhaps you want to tell tasks to end with `shutdownNow()`. In these

scenarios, you wouldn't use the try-with-resources. Table 13.6 describes the behavior of these options.

TABLE 13.6 `ExecutorService` states

Scenario	Description	More tasks allowed	`isShutdown()`	`isTerminated()`
Active	Accepts tasks	Yes	`false`	`false`
`shutdown()`	Runs waiting tasks to completion, but doesn't accept more	No	`true`	`false` while tasks running `true` when tasks complete
`close()`	Calls `shutdown()` and then awaits termination of executing tasks	No	`true`	`true`
`shutdownNow()`	Stops executing tasks and cancels waiting tasks	No	`true`	`false` while tasks running `true` when tasks complete

Figure 13.4 shows how `ExecutorService` separates the states of shutdown and terminated.

When it's winding down and tasks are running, `isShutdown()` can return `true`, while `isTerminated()` can return `false`. Once the tasks complete, both methods return `true`.

Scheduling Tasks

Often in Java, we need to schedule a task to happen at some future time. We might even need to schedule the task to happen repeatedly, at some set interval. For example, imagine that we want to check the supply of food for zoo animals once an hour and fill it as needed.

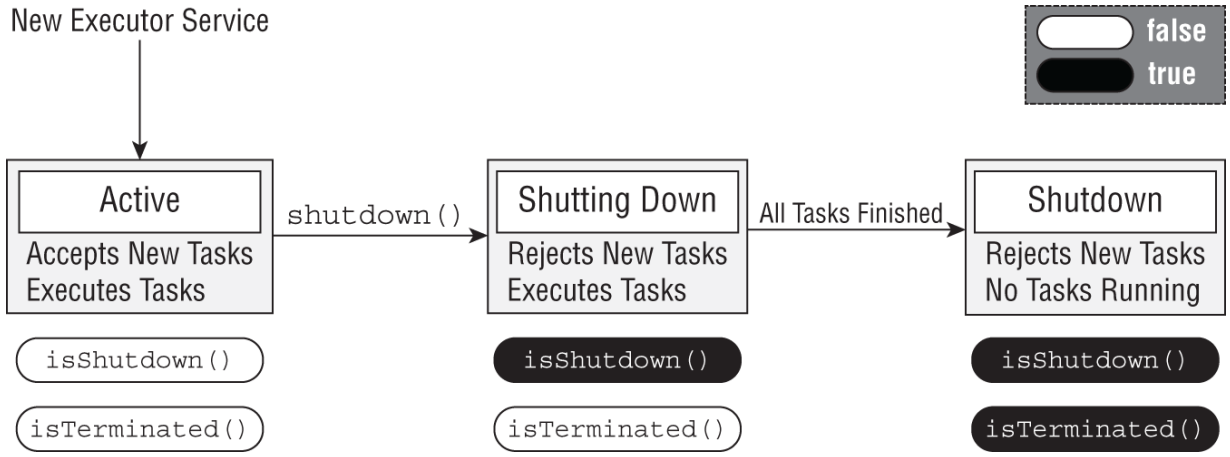

FIGURE 13.4 Thread Executor Lifecycle

`ScheduledExecutorService`, which is a subinterface of `ExecutorService`, can be used for just such a task.

Like `ExecutorService`, we obtain an instance of `ScheduledExecutorService` using a factory method in the `Executors` class, as shown in the following snippet:

```
ScheduledExecutorService service
   = Executors.newSingleThreadScheduledExecutor();
```

Refer to Table 13.7 for our summary of `ScheduledExecutorService` methods. Each of these methods returns a `ScheduledFuture` object.

TABLE 13.7 `ScheduledExecutorService` methods

Method name	Description
`schedule``(Callable<V> callable, long delay, TimeUnit unit)`	Creates and executes `Callable` task after given delay
`schedule``(Runnable command, long delay, TimeUnit unit)`	Creates and executes `Runnable` task after given delay
`scheduleAtFixedRate``(Runnable command, long initialDelay, long period, TimeUnit unit)`	Creates and executes `Runnable` task after given initial delay, creating new task every period value that passes
`scheduleWithFixedDelay``(Runnable command, long initialDelay, long delay, TimeUnit unit)`	Creates and executes `Runnable` task after given initial delay and subsequently with given delay between termination of one execution and commencement of next

In practice, these methods are among the most convenient in the Concurrency API, as they perform relatively complex tasks with a single line of code. The delay and period parameters rely on the `TimeUnit` argument to determine the format of the value, such as seconds or milliseconds.

The first two `schedule()` methods in Table 13.7 take a `Callable` or `Runnable`, respectively; perform the task after some delay; and return a `ScheduledFuture` instance. The `ScheduledFuture` interface is identical to the `Future` interface, except that it includes a `getDelay()` method that returns the remaining delay. The following uses the `schedule()` method with `Callable` and `Runnable` tasks:

```
try (var service =
Executors.newSingleThreadScheduledExecutor()) {
   Runnable task1 = () -> System.out.println("Hello Zoo");
   Callable<String> task2 = () -> "Monkey";
   ScheduledFuture<?> r1 = service.schedule(task1, 10,
TimeUnit.SECONDS);
   ScheduledFuture<?> r2 = service.schedule(task2, 8,
```

```
TimeUnit.MINUTES);
}
```

The first task is scheduled 10 seconds in the future, whereas the second task is scheduled 8 minutes in the future.

While these tasks are scheduled in the future, the actual execution may be delayed. For example, there may be no threads available to perform the tasks, at which point they will just wait in the queue. Also, if the `ScheduledExecutorService` has completely shut down by the time the scheduled task execution time is reached, then these tasks will be discarded.

Each of the `ScheduledExecutorService` methods is important and has real-world applications. For example, you can use the `schedule()` command to check on the state of cleaning a lion's cage. It can then send out notifications if it is not finished or even call `schedule()` to check again later.

The last two methods in Table 13.7 might be a little confusing if you have not seen them before. Conceptually, they are similar as they both perform the same task repeatedly after an initial delay. The difference is related to the timing of the process and when the next task starts.

The `scheduleAtFixedRate()` method creates a new task and submits it to the executor every period, regardless of whether the previous task finished. The following example executes a `Runnable` task every minute, following an initial five-minute delay:

```
service.scheduleAtFixedRate(task1, 5, 1, TimeUnit.MINUTES);
```

The `scheduleAtFixedRate()` method is useful for tasks that need to be run at specific intervals, such as checking the health of the animals once a day. Even if it takes two hours to examine an animal on Monday, this doesn't mean that Tuesday's exam should start any later in the day.

Bad things can happen with `scheduleAtFixedRate()` if each task consistently takes longer to run than the execution interval. Imagine if your boss came by your desk every minute and dropped off a piece of paper. Now imagine that it took you five minutes to read each piece of paper. Before long, you would be drowning in piles of paper. This is how an executor feels. Given enough time, the program would submit more tasks to the executor service than could fit in memory, causing the program to crash.

On the other hand, the `scheduleWithFixedDelay()` method creates a new task only after the previous task has finished. For example, if a task runs at 12:00 and takes five minutes to finish, with a period between executions of two minutes, the next task will start at 12:07.

```
service.scheduleWithFixedDelay(task1, 0, 2, TimeUnit.MINUTES);
```

The `scheduleWithFixedDelay()` method is useful for processes that you want to happen repeatedly but whose specific time is unimportant. For example, imagine that we have a zoo cafeteria worker who periodically restocks the salad bar throughout the day. The process can take 20 minutes or more, since it requires the worker to haul a large number of items from the back room. Once the worker has filled the salad bar with fresh food, they don’t need to check at some specific time, just after enough time has passed for it to become low on stock again.

Increasing Concurrency with Pools

All of our examples up until now have been with a single-thread executor, which, while interesting, weren’t particularly useful. After all, the name of this chapter is “Concurrency,” and you can’t do a lot of that with a single-thread executor!

Luckily, the `Executors` class includes a variety of methods that act on a pool of threads. A *thread pool* is a group of pre-instantiated reusable

threads that are available to perform a set of arbitrary tasks. Table 13.8 includes our two previous single-thread executor methods, along with the new ones that you should know for the exam.

TABLE 13.8 `Executors` factory methods

Method name	Description
`ExecutorService` **`newSingleThreadExecutor`**`()`	Creates single-threaded executor that uses single worker platform thread operating off unbounded queue. Results are processed sequentially in the order in which they are submitted.
`ScheduledExecutorService` **`newSingleThreadScheduledExecutor`**`()`	Creates single-threaded executor for platform threads that can schedule commands to run after given delay or to execute periodically.
`ExecutorService` **`newCachedThreadPool`**`()`	Creates platform thread pool that creates new threads as needed but reuses previously constructed threads when they are available.
`ExecutorService` **`newFixedThreadPool`**`(int)`	Creates platform thread pool that reuses fixed number of threads operating off shared unbounded queue.
`ScheduledExecutorService` **`newScheduledThreadPool`**`(int)`	Creates platform thread pool that can schedule commands to run after given delay or execute periodically.
`ExecutorService` **`newVirtualThreadPerTaskExecutor`**`()`	Creates thread pool that creates a new virtual thread for each task.

As shown in Table 13.8, these methods return the same instance types, `ExecutorService` and `ScheduledExecutorService`, that we used earlier in

this chapter. In other words, all of our previous examples are compatible with these new pooled-thread executors!

The difference between a single-thread and a pooled-thread executor is what happens when a task is already running. While a single-thread executor will wait for the thread to become available before running the next task, a pooled-thread executor can execute the next task concurrently. If the pool runs out of available threads, the task will be queued by the thread executor and wait to be completed.

All but the last row in Table 13.8 are for platform threads. Virtual threads have their own factory method `newVirtualThreadPerTaskExecutor`. They are not pooled since they are so lightweight. This allows a new virtual thread to be used for each task.

Writing Thread-Safe Code

Thread-safety is the property of an object that guarantees safe execution by multiple threads at the same time. Since threads run in a shared environment and memory space, how do we prevent two threads from interfering with each other? We must organize access to data so that we don't end up with invalid or unexpected results.

In this part of the chapter, we show how to use a variety of techniques to protect data, including atomic classes, `synchronized` blocks, the `Lock` framework, and cyclic barriers.

Understanding Thread-Safety

Imagine that our zoo has a program to count sheep, preferably one that won't put the zoo workers to sleep! Each zoo worker runs out to a field, adds a new sheep to the flock, counts the total number of sheep, and runs back to us to report the results. The following shows this conceptually, choosing a thread pool size so that all tasks can be run concurrently:

```
1:  import java.util.concurrent.*;
2:  public class SheepManager {
3:     private int sheepCount = 0;
4:     private void incrementAndReport() {
5:        System.out.print((++sheepCount) + " ");
6:     }
```

```
7:     public static void main(String[] args) {
8:        try (var service = Executors.newFixedThreadPool(20))
{
9:           SheepManager manager = new SheepManager();
10:          for (int i = 0; i < 10; i++)
11:             service.submit(() ->
manager.incrementAndReport());
12:    } } }
```

What does this program output? You might think it will output numbers from `1` to `10`, in order, but that is far from guaranteed. It may output in a different order. Worse yet, it may print some numbers twice and not print some numbers at all! The following are possible outputs of this program:

```
1 2 3 4 5 6 7 8 9 10
1 9 8 7 3 6 6 2 4 5
1 8 7 3 2 6 5 4 2 9
```

So, what went wrong? In this example, we use the pre-increment (++) operator to update the `sheepCount` variable. A problem occurs when two threads both read the "old" value before either thread writes the "new" value of the variable. The two assignments become redundant; they both assign the same new value, with one thread overwriting the results of the other. Figure 13.5 demonstrates this problem with two threads, assuming that `sheepCount` has a starting value of `1`.

You can see in Figure 13.5 that both threads read and write the same values, causing one of the two `++sheepCount` operations to be lost. Therefore, the increment operator `++` is not thread-safe. As you will see later in this chapter, the unexpected result of two or more tasks executing at the same time is referred to as a *race condition*.

Conceptually, the idea here is that some zoo workers may run faster on their way to the field but more slowly on their way back and report late. Other workers may get to the field last but somehow be the first ones back to report the results.

The `volatile` modifier can be used on an instance variable to ensure a thread does not see any intermediary values while an operation is being performed. Unfortunately, adding it to the `sheepCount` variable in our previous example is insufficient for thread-safety since `++sheepCount` is really two separate read and write operations. In practice, `volatile` is rarely used. We only mention it because it has been known to show up on the exam from time to time.

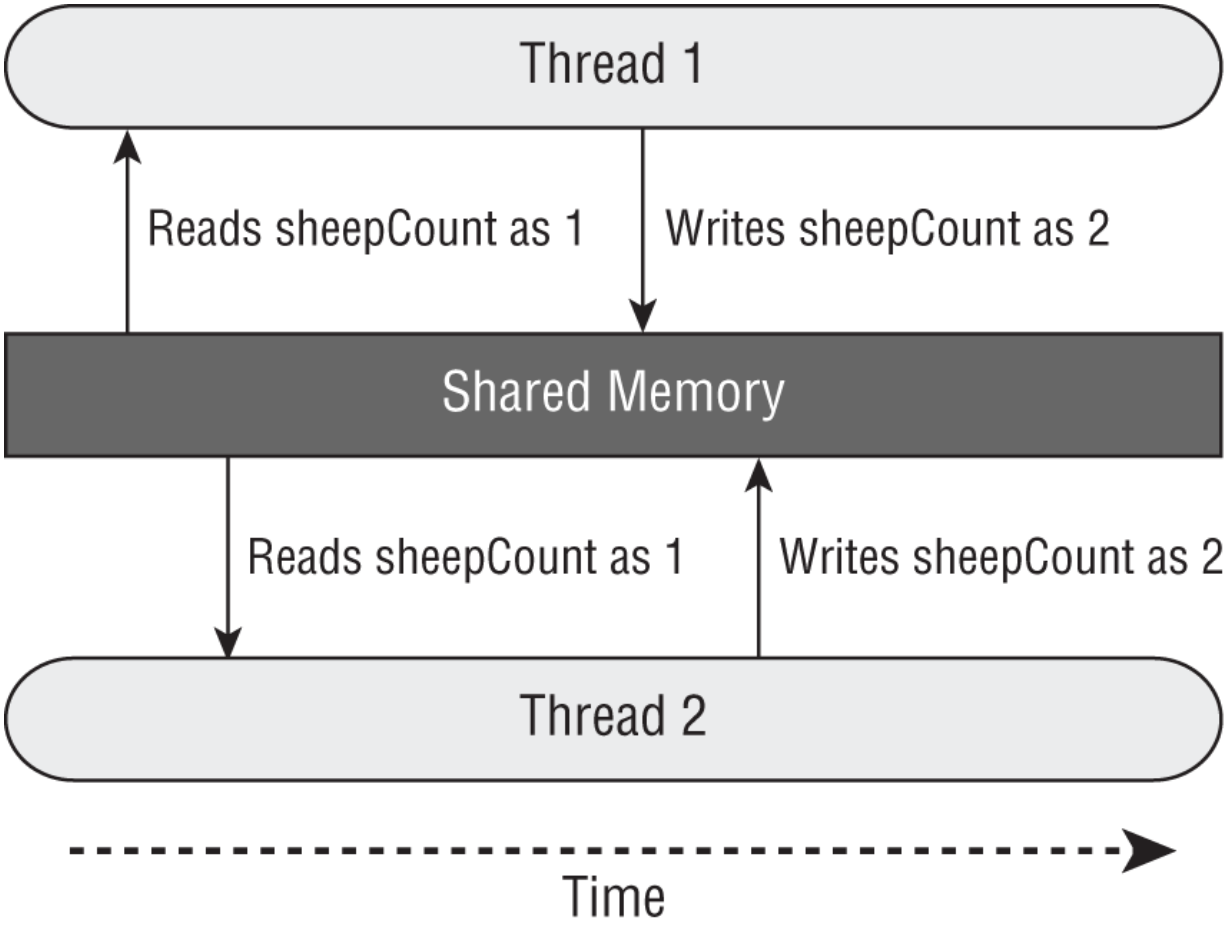

FIGURE 13.5 Lack of thread synchronization

Protecting Data with Atomic Classes

In our previous `SheepManager` application, the same values were sometimes printed twice, with the highest counter being `9` instead of `10`. As we saw, the increment operator `++` is not thread-safe because the operation is not atomic, carrying out two tasks, read and write, that can be interrupted by other threads.

Atomic is the property of an operation to be carried out as a single unit of execution without any interference from another thread. A thread-safe atomic version of the increment operator would perform the read and write of the variable as a single operation, not allowing any other threads to access the variable during the operation. Figure 13.6 shows the result of making the `sheepCount` variable atomic.

In this case, any thread trying to access the `sheepCount` variable while an atomic operation is in process will have to wait until the atomic operation on the variable is complete. Conceptually, this is like setting a rule for our zoo workers that there can be only one employee in the field at a time, although they may not each report their results in order.

Since accessing primitives and references is common in Java, the Concurrency API includes numerous useful classes in the `java.util.concurrent.atomic` package. Table 13.9 lists the atomic classes with which you should be familiar for the exam. As with many of the classes in the Concurrency API, these classes exist to make your life easier.

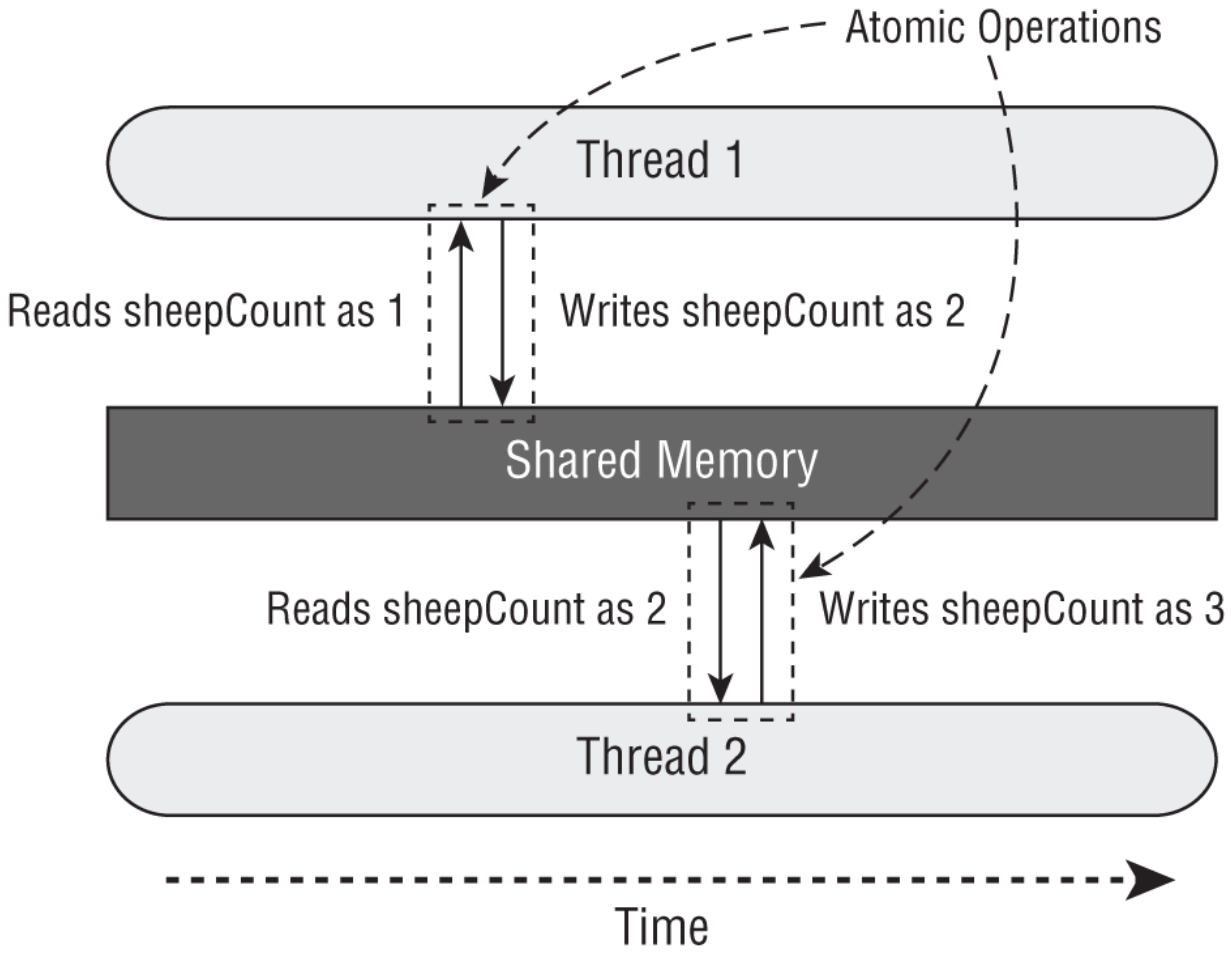

FIGURE 13.6 Thread synchronization using atomic operations

TABLE 13.9 Atomic classes

Class name	Description
`AtomicBoolean`	A `boolean` value that may be updated atomically
`AtomicInteger`	An `int` value that may be updated atomically
`AtomicLong`	A `long` value that may be updated atomically

How do we use an atomic class? Each class includes numerous methods that are equivalent to many of the built-in operators that we use on primitives, such as the assignment operator (=) and the increment operators (++). We describe the common atomic methods that you should know for the exam in Table 13.10. The *`type`* is determined by the class.

TABLE 13.10 Common atomic methods

Method name	Description
`get()`	Retrieves current value
`set(`*`type`* `newValue)`	Sets given value, equivalent to assignment `=` operator
`getAndSet(`*`type`* `newValue)`	Atomically sets new value and returns old value
`incrementAndGet()`	For numeric classes, atomic pre-increment operation equivalent to `++value`
`getAndIncrement()`	For numeric classes, atomic post-increment operation equivalent to `value++`
`decrementAndGet()`	For numeric classes, atomic pre-decrement operation equivalent to `--value`
`getAndDecrement()`	For numeric classes, atomic post-decrement operation equivalent to `value--`

In the following example, assume we import the `atomic` package and then update our `SheepManager` class with an `AtomicInteger`:

```
3:    private AtomicInteger sheepCount = new AtomicInteger(0);
4:    private void incrementAndReport() {
5:       System.out.print(sheepCount.incrementAndGet() + " ");
6:    }
```

How does this implementation differ from our previous examples? When we run this modification, we get varying output, such as the following:

```
2 3 1 4 5 6 7 8 9 10
1 4 3 2 5 6 7 8 9 10
1 4 3 5 6 2 7 8 10 9
```

Unlike our previous sample output, the numbers `1` through `10` will always be printed, although the order is still not guaranteed. Don’t worry; we address that issue shortly. The key in this section is that using the atomic classes ensures that the data is consistent between workers and that no values are lost due to concurrent modifications.

Improving Access with *synchronized* Blocks

While atomic classes are great at protecting a single variable, they aren't particularly useful if you need to execute a series of commands or call a method. For example, we can't use them to update two atomic variables at the same time. How do we improve the results so that each worker is able to increment and report the results in order?

The most common technique is to use a monitor to synchronize access. A *monitor*, also called a *lock*, is a structure that supports *mutual exclusion*, which is the property that at most one thread is executing a particular segment of code at a given time.

In Java, any `Object` can be used as a monitor, along with the `synchronized` keyword, as shown in the following example:

```
var manager = new SheepManager();
synchronized(manager) {
   // Work to be completed by one thread at a time
}
```

This example is referred to as a *synchronized block*. Each thread that arrives will first check if any threads are already running the block. If the lock is not available, the thread will transition to a `BLOCKED` state until it can "acquire the lock." If the lock is available (or the thread already holds the lock), the single thread will enter the block, preventing all other threads from entering. Once the thread finishes executing the block, it will release the lock, allowing one of the waiting threads to proceed.

To synchronize access across multiple threads, each thread must have access to the *same* `Object`. If each thread synchronizes on different objects, the code is not thread-safe.

Let's revisit our `SheepManager` example that used `++sheepCount` and see whether we can improve the results so that each worker increments and

outputs the counter in order. Let's say that we replaced our `for()` loop with the following implementation:

```
10: for (int i = 0; i < 10; i++) {
11:    synchronized(manager) {
12:       service.submit(() -> manager.incrementAndReport());
13:    }
14: }
```

Does this solution fix the problem? No, it does not! Can you spot the problem? We've synchronized the *creation* of the threads but not the *execution* of the threads. In this example, the threads would be created one at a time, but they might all still execute and perform their work simultaneously, resulting in the same type of output that you saw earlier. We did say diagnosing and resolving thread problems is difficult in practice!

This is a corrected version of the `SheepManager` class that orders the workers:

```
1:  import java.util.concurrent.*;
2:  public class SheepManager {
3:     private int sheepCount = 0;
4:     private void incrementAndReport() {
5:        synchronized(this) {
6:           System.out.print((++sheepCount) + " ");
7:        }
8:     }
9:     public static void main(String[] args) {
10:       try (var service = Executors.newFixedThreadPool(20))
{
11:          var manager = new SheepManager();
12:          for (int i = 0; i < 10; i++)
13:             service.submit(() ->
manager.incrementAndReport());
14:       } } }
```

When this code executes, it will consistently output the following:

```
1 2 3 4 5 6 7 8 9 10
```

Although all threads are still created and executed at the same time, they each wait at the `synchronized` block for the worker to increment and report the result before entering. In this manner, each zoo worker waits for the previous zoo worker to come back before running out on the field. While

it’s random which zoo worker will run out next, it is guaranteed that there will be at most one on the field and that the results will be reported in order.

We could have synchronized on any object, as long as it was the same object. For example, the following code snippet would also work:

```
4:    private final Object herd = new Object();
5:    private void incrementAndReport() {
6:       synchronized(herd) {
7:          System.out.print((++sheepCount) + " ");
8:       }
9:    }
```

Although we didn’t need to make the `herd` variable `final`, doing so ensures that it is not reassigned after threads start using it.

Synchronizing Methods

In the previous example, we established our monitor using `synchronized(this)` around the body of the method. Java provides a more convenient syntax for doing so. We can add the `synchronized` modifier to any instance method to synchronize automatically on the object itself. For example, the following two method definitions are equivalent:

```
void sing() {
   synchronized(this) {
      System.out.print("La la la!");
   }
}
synchronized void sing() {
   System.out.print("La la la!");
}
```

The first uses a `synchronized` block, whereas the second uses the `synchronized` method modifier. Which you use is completely up to you.

We can also apply the `synchronized` modifier to `static` methods. What object is used as the monitor when we synchronize on a `static` method? The class object, of course! For example, the following two methods are equivalent for `static` synchronization inside our `SheepManager` class:

```
static void dance() {
   synchronized(SheepManager.class) {
      System.out.print("Time to dance!");
```

```
   }
}
static synchronized void dance() {
   System.out.print("Time to dance!");
}
```

As before, the first uses a `synchronized` block, with the second example using the `synchronized` modifier. You can use `static` synchronization if you need to order thread access across all instances rather than a single instance.

Understanding the Lock Framework

A `synchronized` block supports only a limited set of functionality. For example, what if we want to check whether a lock is available and, if it is not, perform some other task? Furthermore, if the lock is never available and we synchronize on it, we might wait forever.

The Concurrency API includes the `Lock` interface, which is conceptually similar to using the `synchronized` keyword but with a lot more bells and whistles.

Applying a *ReentrantLock*

The `Lock` interface is pretty easy to use. When you need to protect a piece of code from multithreaded processing, create an instance of `Lock` that all threads have access to. Each thread then calls `lock()` before it enters the protected code and calls `unlock()` before it exits the protected code.

For contrast, the following shows two implementations, one with a `synchronized` block and one with a `Lock` instance. While longer, the `Lock` solution has a number of features not available to the `synchronized` block.

```
// Implementation #1 with a synchronized block
var object = new Object();
synchronized(object) {
   // Protected code
}

// Implementation #2 with a Lock
var myLock = new ReentrantLock();
try {
   myLock.lock();
```

```
   // Protected code
} finally {
   myLock.unlock();
}
```

These two implementations are conceptually equivalent. The `ReentrantLock` class is a simple monitor that implements the `Lock` interface and supports mutual exclusion. In other words, at most one thread is allowed to hold a lock at any given time.

While certainly not required, it is a good practice to use a `try`/`finally` block with `Lock` instances to ensure that any acquired locks are properly released. This can help prevent a resource leak in practice.

The `ReentrantLock` class ensures that once a thread has called `lock()` and obtained the lock, all other threads that call `lock()` will wait until the first thread calls `unlock()`. Which thread gets the lock next depends on the parameters used to create the `Lock` object.

If you need to control the order threads run, you can use the `ReentrantLock` constructor that takes a single `boolean` "fairness" parameter. In practice, you should enable fairness only when ordering is absolutely required, as it could lead to a significant slowdown.

Besides always making sure to release a lock, you also need to be sure that you only release a lock that you have. If you attempt to release a lock that you do not have, you will get an exception at runtime.

```
var lock = new ReentrantLock();
lock.unlock();  // IllegalMonitorStateException
```

The `Lock` interface includes four methods you should know for the exam, as listed in Table 13.11.

TABLE 13.11 `Lock` methods

Method name	Description
`void` **`lock`**`()`	Requests lock and blocks until lock is acquired.
`void` **`unlock`**`()`	Releases lock.
`boolean` **`tryLock`**`()`	Requests lock and returns immediately. Returns `boolean` indicating whether lock was successfully acquired.
`boolean` **`tryLock`**`(long timeout, TimeUnit unit)`	Requests lock and blocks for specified time or until lock is acquired. Returns `boolean` indicating whether lock was successfully acquired.

Attempting to Acquire a Lock

While the `ReentrantLock` class allows you to wait for a lock, it so far suffers from the same problem as a `synchronized` block. A thread could end up waiting forever to obtain a lock. Luckily, Table 13.11 includes two additional methods that make the `Lock` interface a lot safer to use than a `synchronized` block.

For convenience, we use the following `printHello()` method for the code in this section:

```
static void printHello(Lock myLock) {
   try {
      myLock.lock();
      System.out.println("Hello");
   } finally {
      myLock.unlock();
   } }
```

The `tryLock()` method will attempt to acquire a lock and immediately return a `boolean` result indicating whether the lock was obtained. Unlike the `lock()` method, it does not wait if another thread already holds the lock. It returns immediately, regardless of whether a lock is available.

The following is a sample implementation using the `tryLock()` method:

```
var myLock = new ReentrantLock();
Thread.ofPlatform().start(() -> printHello(myLock));
if (myLock.tryLock()) {
   try {
      System.out.println("Lock obtained, entering protected
code");
   } finally {
      myLock.unlock();
   }
} else {
   System.out.println("Unable to acquire lock, doing something
else");
}
```

When you run this code, it could produce either the `if` or `else` message, depending on the order of execution. It will always print `Hello`, though, as the call to `lock()` in `printHello()` will wait indefinitely for the lock to become available. A fun exercise is to insert some `Thread.sleep()` delays into this snippet to encourage a particular message to be displayed.

Like `lock()`, the `tryLock()` method should be used with a `try`/`finally` block. Fortunately, you need to release the lock only if it was successfully acquired. For this reason, it is common to use the output of `tryLock()` in an `if` statement, so that `unlock()` is called only when the lock is obtained.

It is imperative that your program always check the return value of the `tryLock()` method. It tells your program whether it is safe to proceed with the operation and whether the lock needs to be released later.

The `Lock` interface includes an overloaded version of `tryLock(long,TimeUnit)` that acts like a hybrid of `lock()` and `tryLock()`. Like the other two methods, if a lock is available, it will immediately return with it. If a lock is unavailable, though, it will wait up to the specified time limit for the lock.

Acquiring the Same Lock Twice

The `ReentrantLock` class maintains a counter of the number of times a lock has been successfully granted to a thread. To release the lock for other threads to use, `unlock()` must be called the same number of times the lock was granted. The following code snippet contains an error. Can you spot it?

```
var myLock = new ReentrantLock();
if (myLock.tryLock()) {
   try {
      myLock.lock();
      System.out.println("Lock obtained, entering protected
code");
   } finally {
      myLock.unlock();
   } }
```

The thread obtains the lock twice but releases it only once. You can verify this by spawning a new thread after this code runs that attempts to obtain a lock. The following prints `false`:

```
Thread.ofPlatform().start(() ->
System.out.print(myLock.tryLock()));
```

It is critical that you release a lock the same number of times it is acquired! For calls with `tryLock()`, you need to call `unlock()` only if the method returned `true`.

Reviewing the *Lock* Framework

To review, the `ReentrantLock` class supports the same features as a `synchronized` block while adding a number of improvements:

- Ability to request a lock without blocking.
- Ability to request a lock while blocking for a specified amount of time.
- A lock can be created with a `fairness` property, in which the lock is granted to threads in the order in which it was requested.

Introducing *ReentrantReadWriteLock*

When working with shared data, reading data is often far more common than writing data. For example, a single operator at the zoo might be updating the lunch menu that thousands of patrons are reading from their phone.

For this reason, `ReentrantReadWriteLock` is a really useful class. It includes separate locks for reading and writing data. At runtime, only one thread can hold the write lock at a time, but many threads can hold the read lock. Having separate locks can help you maximize concurrent access.

```
var lock = new ReentrantReadWriteLock();
lock.writeLock().lock();
lock.readLock().lock();
System.out.println(lock.isWriteLocked());     // true
System.out.println(lock.getReadLockCount());  // 1

lock.writeLock().unlock();
System.out.println(lock.isWriteLocked());     // false
System.out.println(lock.getReadLockCount());  // 1

lock.readLock().unlock();
System.out.println(lock.getReadLockCount());  // 0
```

On the exam, you are likely to see it used within a single class. You need to know that you can trivially get a read lock after acquiring a write lock because reading is a subset of write. However, if you attempt to get the read lock first, the code will hang as you can't upgrade to a write lock.

```
var lock = new ReentrantReadWriteLock();
lock.readLock().lock();
lock.writeLock().lock();  // Wait forever
```

Orchestrating Tasks with a *CyclicBarrier*

We started the thread-safety topic by discussing protecting individual variables and then moved on to blocks of code and locks. We complete our

discussion of thread-safety by showing how to orchestrate complex tasks with many steps.

Our zoo workers are back, and this time they are cleaning pens. Imagine a lion pen that needs to be emptied, cleaned, and then refilled with the lions. To complete the task, we have assigned four zoo workers. Obviously, we don’t want to start cleaning the cage while a lion is roaming in it, lest we end up losing a zoo worker! Furthermore, we don’t want to let the lions back into the pen while it is still being cleaned.

We could have all of the work completed by a single worker, but this would be slow and ignore the fact that we have three zoo workers standing by to help. A better solution would be to have all four zoo employees work concurrently, pausing between the end of one set of tasks and the start of the next.

To coordinate these tasks, we can use the `CyclicBarrier` class:

```
import java.util.concurrent.*;
public class LionPenManager {
   private void removeLions() { System.out.println("Removing
lions");   }
   private void cleanPen()    { System.out.println("Cleaning
the pen"); }
   private void addLions()    { System.out.println("Adding
lions");     }
   public void performTask() {
      removeLions();
      cleanPen();
      addLions();
   }
   public static void main(String[] args) {
      try (var service = Executors.newFixedThreadPool(4)) {
         var manager = new LionPenManager();
         for (int i = 0; i < 4; i++)
            service.submit(() -> manager.performTask());
      } } }
```

The following is sample output based on this implementation:

```
Removing lions
Removing lions
Cleaning the pen
Adding lions
Removing lions
```

```
Cleaning the pen
Adding lions
Removing lions
Cleaning the pen
Adding lions
Cleaning the pen
Adding lions
```

Although the results are ordered within a single thread, the output is entirely random among multiple workers. We see that some lions are still being removed while the cage is being cleaned, and other lions are added before the cleaning process is finished. Let's hope none of the zoo workers get eaten!

We can improve these results by using the `CyclicBarrier` class. The `CyclicBarrier` class takes in its constructors a `limit` value, indicating the number of threads to wait for. As each thread finishes, it calls the `await()` method on the cyclic barrier. Once the specified number of threads have each called `await()`, the barrier is released, and all threads can continue.

```
import java.util.concurrent.*;
public class LionPenManager {
   private void removeLions() { System.out.println("Removing
lions");   }
   private void cleanPen()    { System.out.println("Cleaning
the pen"); }
   private void addLions()    { System.out.println("Adding
lions");     }
   public void performTask(CyclicBarrier c1, CyclicBarrier c2)
{
      try {
         removeLions();
         c1.await();
         cleanPen();
         c2.await();
         addLions();
      } catch (InterruptedException | BrokenBarrierException e)
{
         // Handle checked exceptions here
      }
   }
   public static void main(String[] args) {
      try (var service = Executors.newFixedThreadPool(4)) {
         var manager = new LionPenManager();
         var c1 = new CyclicBarrier(4);
```

```
         var c2 = new CyclicBarrier(4,
            () -> System.out.println("*** Pen Cleaned!"));
         for (int i = 0; i < 4; i++)
            service.submit(() -> manager.performTask(c1, c2));
      } } }
```

The following is sample output based on this revised implementation of our `LionPenManager` class:

```
Removing lions
Removing lions
Removing lions
Removing lions
Cleaning the pen
Cleaning the pen
Cleaning the pen
Cleaning the pen
*** Pen Cleaned!
Adding lions
Adding lions
Adding lions
Adding lions
```

As you can see, all of the results are now organized. Removing the lions happens in one step, as does cleaning the pen and adding the lions back in. In this example, we used two different constructors for our `CyclicBarrier` objects, the latter of which executes a `Runnable` instance upon completion.

The `CyclicBarrier` class allows us to perform complex, multithreaded tasks while all threads stop and wait at logical barriers. This solution is superior to a single-threaded solution, as the individual tasks, such as removing the lions, can be completed in parallel by all four zoo workers.

Reusing *CyclicBarrier*

After a `CyclicBarrier` limit is reached (aka the barrier is broken), all threads are released, and the number of threads waiting on the `CyclicBarrier` goes back to zero. At this point, the `CyclicBarrier` may be used again for a new set of waiting threads. For example, if our `CyclicBarrier` limit is 5 and we have 15 threads that call `await()`, the `CyclicBarrier` will be activated a total of three times.

Using Concurrent Collections

Besides managing threads, the Concurrency API includes interfaces and classes that help you coordinate access to collections shared by multiple tasks. By collections, we are of course referring to the Java Collections Framework that we introduced in Chapter 9, "Collections and Generics." In this section, we demonstrate many of the concurrent classes available to you when using the Concurrency API.

Understanding Memory Consistency Errors

The purpose of the concurrent collection classes is to solve common memory consistency errors. A *memory consistency error* occurs when two threads have inconsistent views of what should be the same data. Conceptually, we want writes on one thread to be available to another thread if it accesses the concurrent collection after the write has occurred.

When two threads try to modify the same nonconcurrent collection, the JVM may throw a `ConcurrentModificationException` at runtime. In fact, it can happen with a single thread. Take a look at the following code snippet:

```
11: var foodData = new HashMap<String, Integer>();
12: foodData.put("penguin", 1);
13: foodData.put("flamingo", 2);
14: for (String key : foodData.keySet())
15:    foodData.remove(key);
```

This snippet will throw a `ConcurrentModificationException` during the second iteration of the loop, since the iterator on `keySet()` is not properly updated after the first element is removed. Changing the first line to use a `ConcurrentHashMap` will prevent the code from throwing an exception at runtime.

```
11: var foodData = new ConcurrentHashMap<String, Integer>();
```

Although we don't usually modify a loop variable, this example highlights the fact that the `ConcurrentHashMap` is ordering read/write access such that all access to the class is consistent. In this code snippet, the iterator created by `keySet()` is updated as soon as an object is removed from the `Map`.

The concurrent classes were created to help avoid common issues in which multiple threads are adding and removing objects from the same collections. At any given instance, all threads should have the same consistent view of the structure of the collection.

Working with Concurrent Classes

You should use a concurrent collection class any time you have multiple threads modify a collection outside a `synchronized` block or method, even if you don't expect a concurrency problem. Without the concurrent collections, multiple threads accessing a collection could result in an exception being thrown or, worse, corrupt data!

If the collection is immutable (and contains immutable objects), the concurrent collections are not necessary. Immutable objects can be accessed by any number of threads and do not require synchronization. By definition, they do not change, so there is no chance of a memory consistency error.

Table 13.12 lists the common concurrent classes with which you should be familiar for the exam.

TABLE 13.12 Concurrent collection classes

Class name	Java Collections interfaces	Sorted?	Blocking?
`ConcurrentHashMap`	`Map` `ConcurrentMap`	No	No
`ConcurrentLinkedQueue`	`Queue`	No	No

`ConcurrentSkipListMap`	`Map`
	`SequencedMap`
	`SortedMap`
	`NavigableMap`
	`ConcurrentMap`
	`ConcurrentNavigableMap`
		Yes	No

`ConcurrentSkipListSet`	`Set`
	`SequencedSet`
	`SortedSet`
	`NavigableSet`
		Yes	No

`CopyOnWriteArrayList`	`List` `SequencedCollection`	No	No
`CopyOnWriteArraySet`	`Set`	No	No
`LinkedBlockingQueue`	`Queue` `BlockingQueue`	No	Yes

Most of the classes in Table 13.12 are just concurrent versions of their nonconcurrent counterpart classes, such as `ConcurrentHashMap` vs. `HashMap`, or `ConcurrentLinkedQueue` vs. `LinkedList`. For the exam, you don't need to know any class-specific concurrent methods. You just need to know the inherited methods, such as `get()` and `set()` for `List` instances.

The `Skip` classes might sound strange, but they are just "sorted" versions of the associated concurrent collections. When you see a class with `Skip` in the name, just think "sorted concurrent" collections, and the rest should follow naturally.

The `CopyOnWrite` classes behave a little differently than the other concurrent data structures you have seen. These classes create a copy of the collection any time a reference is added, removed, or changed in the

collection and then update the original collection reference to point to the copy. These classes are commonly used to ensure an iterator doesn't see modifications to the collection.

Let's take a look at how this works with an example:

```
List<Integer> favNumbers = new CopyOnWriteArrayList<>
(List.of(4, 3, 42));
for (var n : favNumbers) {
   System.out.print(n + " ");  // 4 3 42
   favNumbers.add(n + 1);
}
System.out.println();
System.out.println("Size: " + favNumbers.size());  // Size: 6
```

Despite adding elements, the iterator is not modified, and the loop executes exactly three times. Alternatively, if we had used a regular `ArrayList` object, a `ConcurrentModificationException` would have been thrown at runtime. The `CopyOnWrite` classes can use a lot of memory, since a new collection structure is created any time the collection is modified. Therefore, they are commonly used in multithreaded environment situations where reads are far more common than writes.

Finally, Table 13.12 includes `LinkedBlockingQueue`, which implements the concurrent `BlockingQueue` interface. This class is just like a regular `Queue`, except that it includes overloaded versions of `offer()` and `poll()` that take a timeout. These methods wait (or block) up to a specific amount of time to complete an operation.

Obtaining Synchronized Collections

Besides the concurrent collection classes that we have covered, the Concurrency API also includes methods for obtaining synchronized versions of existing nonconcurrent collection objects. These synchronized methods are defined in the `Collections` class. They operate on the inputted collection and return a reference that is the same type as the underlying collection. We list these `static` methods in Table 13.13.

TABLE 13.13 Synchronized `Collections` methods

synchronizedCollection(Collection<T> c)
synchronizedList(List<T> list)
synchronizedMap(Map<K,V> m)
synchronizedNavigableMap(NavigableMap<K,V> m)
synchronizedNavigableSet(NavigableSet<T> s)
synchronizedSet(Set<T> s)
synchronizedSortedMap(SortedMap<K,V> m)
synchronizedSortedSet(SortedSet<T> s)

If you're writing code to create a collection and it requires synchronization, you should use the classes defined in Table 13.12. On the other hand, if you are passed a nonconcurrent collection and need synchronization, use the methods in Table 13.13.

Identifying Threading Problems

Now that you know how to write thread-safe code, let's talk about what qualifies as a threading problem. A threading problem can occur in multithreaded applications when two or more threads interact in an unexpected and undesirable way. For example, two threads may block each other from accessing a particular segment of code.

The Concurrency API was created to help eliminate potential threading issues common to all developers. As you have seen, the Concurrency API creates threads and manages complex thread interactions for you, often in just a few lines of code.

Although the Concurrency API reduces the potential for threading issues, it does not eliminate them. In practice, finding and identifying threading issues within an application is often one of the most difficult tasks a developer can undertake.

Understanding Liveness

As you have seen in this chapter, many thread operations can be performed independently, but some require coordination. For example, synchronizing on a method requires all threads that call the method to wait for other threads to finish before continuing. You also saw earlier in the chapter that threads in a `CyclicBarrier` will each wait for the barrier limit to be reached before continuing.

What happens to the application while all of these threads are waiting? In many cases, the waiting is ephemeral, and the user has very little idea that any delay has occurred. In other cases, though, the waiting may be extremely long, perhaps infinite.

Liveness is the ability of an application to be able to execute in a timely manner. Liveness problems, then, are those in which the application becomes unresponsive or is in some kind of "stuck" state. More precisely, liveness problems are often the result of a thread entering a `BLOCKING` or `WAITING` state forever, or repeatedly entering/exiting these states. For the exam, there are three types of liveness issues with which you should be familiar: deadlock, starvation, and livelock.

Deadlock

Deadlock occurs when two or more threads are blocked forever, each waiting on the other. We can illustrate this principle with the following example. Imagine that our zoo has two foxes: Foxy and Tails. Foxy likes to eat first and then drink water, while Tails likes to drink water first and then eat. Furthermore, neither animal likes to share, and they will finish their meal only if they have exclusive access to both food and water.

The zookeeper places the food on one side of the environment and the water on the other side. Although our foxes are fast, it still takes them 100 milliseconds to run from one side of the environment to the other.

What happens if Foxy gets the food first and Tails gets the water first? The following application models this behavior:

```
class Food {}
class Water {}
public record Fox(String name) {
   public void eatAndDrink(Food food, Water water) {
      synchronized(food) {
         System.out.println(name() + " Got Food!");
```

```
            move();
            synchronized(water) {
               System.out.println(name() + " Got Water!");
            } } }
   public void drinkAndEat(Food food, Water water) {
      synchronized(water) {
            System.out.println(name() + " Got Water!");
            move();
            synchronized(food) {
               System.out.println(name() + " Got Food!");
            } } }
   public void move() {
      try { Thread.sleep(100); } catch (InterruptedException e)
{}
   }
   public static void main(String[] args) {
      // Create participants and resources
      var foxy = new Fox("Foxy");
      var tails = new Fox("Tails");
      var food = new Food();
      var water = new Water();
      // Process data
      try (var service = Executors.newScheduledThreadPool(10))
{
            service.submit(() -> foxy.eatAndDrink(food,water));
            service.submit(() -> tails.drinkAndEat(food,water));
      } } }
```

In this example, Foxy obtains the food and then moves to the other side of the environment to obtain the water. Unfortunately, Tails already drank the water and is waiting for the food to become available. The result is that our program outputs the following, and it hangs indefinitely:

```
Foxy Got Food!
Tails Got Water!
```

This example is considered a deadlock because both participants are permanently blocked, waiting on resources that will never become available.

Starvation

Starvation occurs when a single thread is perpetually denied access to a shared resource or lock. The thread is still active, but it is unable to

complete its work as a result of other threads constantly taking the resource that it is trying to access.

In our fox example, imagine that we have a pack of very hungry, very competitive foxes in our environment. Every time Foxy stands up to go get food, one of the other foxes sees her and rushes to eat before her. Foxy is free to roam around the enclosure, take a nap, and howl for a zookeeper but is never able to obtain access to the food. In this example, Foxy literally and figuratively experiences starvation. It's a good thing that this is just a theoretical example!

Livelock

Livelock occurs when two or more threads are conceptually blocked forever, although they are each still active and trying to complete their task. Livelock is a special case of resource starvation in which two or more threads actively try to acquire a set of locks, are unable to do so, and restart part of the process.

Livelock is often a result of two threads trying to resolve a deadlock. Returning to our fox example, imagine that Foxy and Tails are both holding their food and water resources, respectively. They each realize that they cannot finish their meal in this state, so they both let go of their food and water, run to the opposite side of the environment, and pick up the other resource. Now Foxy has the water, Tails has the food, and neither is able to finish their meal!

If Foxy and Tails continue this process forever, it is referred to as *livelock*. Both Foxy and Tails are active, running back and forth across their area, but neither can finish their meal. Foxy and Tails are executing a form of failed deadlock recovery. Each fox notices that they are potentially entering a deadlock state and responds by releasing all of its locked resources. Unfortunately, the lock and unlock process is cyclical, and the two foxes are conceptually deadlocked.

In practice, livelock is often a difficult issue to detect. Threads in a livelock state appear active and able to respond to requests, even when they are stuck in an endless cycle.

Managing Race Conditions

A *race condition* is an undesirable result that occurs when two tasks that should be completed sequentially are completed at the same time. We encountered examples of race conditions earlier in the chapter when we introduced synchronization.

While Figure 13.5 shows a classical thread-based example of a race condition, we now provide a more illustrative example. Imagine that two zoo patrons, Olivia and Sophia, are signing up for an account on the zoo's new visitor website. Both of them want to use the same username, `ZooFan`, and each sends a request to create the account at the same time, as shown in Figure 13.7.

What result does the web server return when both users attempt to create an account with the same username in Figure 13.7?


An image illustrates the classical thread of a race conditions. Two zoo patrons, Olivia and Sophia, are signing up to zoo web server using a same username of zoofan.

FIGURE 13.7 Race condition on user creation

Possible Outcomes for This Race Condition

- Both users are able to create accounts with the username `ZooFan`.
- Neither user is able to create an account with the username `ZooFan`, and an error message is returned to both users.
- One user is able to create an account with the username `ZooFan`, while the other user receives an error message.

The first outcome is *really bad*, as it leads to users trying to log in with the same username. Whose data do they see when they log in? The second outcome causes both users to have to try again, which is frustrating but at least doesn't lead to corrupt or bad data.

The third outcome is often considered the best solution. Like the second situation, we preserve data integrity; but unlike the second situation, at least one user is able to move forward on the first request, avoiding additional race condition scenarios.

For the exam, you should understand that race conditions lead to invalid data if they are not properly handled. Even the solution where both participants fail to proceed is preferable to one in which invalid data is permitted to enter the system.

Working with Parallel Streams

We conclude this chapter by combining what you learned in Chapter 10, "Streams," with the concepts you learned about in this chapter. One of the most powerful features of the Stream API is built-in concurrency support. Up until now, all of the streams you have worked with have been serial streams. A *serial stream* is a stream in which the results are ordered, with only one entry being processed at a time.

A *parallel stream* is capable of processing results concurrently, using multiple threads. For example, you can use a parallel stream and the `map()` operation to operate concurrently on the elements in the stream, vastly improving performance over processing a single element at a time.

Using a parallel stream can change not only the performance of your application but also the expected results. As you shall see, some operations also require special handling to be able to be processed in a parallel manner.

Generating Random Numbers

In Chapter 4, "Core APIs," you learned about generating random numbers for a single-threaded program. To generate random numbers in a multithreaded program, you use the `ThreadLocalRandom` class instead. To start out, you get an instance using `current()`. Then all the instance methods are available to you.

```
ThreadLocalRandom.current()
   .ints()
   .limit(5)
   .forEach(System.out::println);  // Prints 5 random ints
```

There are six methods available for working with ints:

- **`ints()`**: Infinite stream of any `int` values
- **`ints(lowestInclusive, highestExclusive)`:** Unlimited stream of `int` values between the two parameters, excluding the second one
- **`ints(numberIntsToInclude)`:** Finite stream of the requested number of `int` values
- **`ints(numberIntsToInclude, lowestInclusive, highestExclusive)`:** Finite stream of the requested number of `int` values between the two parameters, excluding the second one
- **`nextInt(highestExclusive)`:** Single `int` between 0 and the parameter, not including the parameter
- **`nextInt(lowestInclusive, highestExclusive)`:** Single `int` between the first parameter and the second parameter, not including the second parameter

Similar methods are available for doubles and longs, such `doubles()`, `nextDouble()`, `longs()`, `nextLong()`, etc.

Creating Parallel Streams

The Stream API was designed to make creating parallel streams quite easy. For the exam, you should be familiar with two ways of creating a parallel stream.

```
Collection<Integer> collection = List.of(1, 2);

Stream<Integer> p1 = collection.stream().parallel();
Stream<Integer> p2 = collection.parallelStream();
```

The first way to create a parallel stream is from an existing stream. Isn't this cool? Any stream can be made parallel! The second way to create a parallel stream is from a Java `Collection` class. We use both of these methods throughout this section.

The `Stream` interface includes a method `isParallel()` that can be used to test whether the instance of a stream supports parallel processing. Some operations on streams preserve the parallel attribute, while others do not.

Performing a Parallel Decomposition

A *parallel decomposition* is the process of taking a task, breaking it into smaller pieces that can be performed concurrently, and then reassembling the results. The more concurrent a decomposition, the greater the performance improvement of using parallel streams.

Let's try it out. First, let's define a reusable function that "does work" just by waiting for five seconds.

```
private static int doWork(int input) {
   try {
      Thread.sleep(5000);
   } catch (InterruptedException e) {}
   return input;
}
```

We can pretend that in a real application, this work might involve calling a database or reading a file. Now let's use this method with a serial stream.

```
10: long start = System.currentTimeMillis();
11: List.of(1, 2, 3, 4, 5)
12:    .stream()
13:    .map(w -> doWork(w))
14:    .forEach(s -> System.out.print(s + " "));
15:
16: System.out.println();
17: var timeTaken = (System.currentTimeMillis()-start)/1000;
18: System.out.println("Time: " + timeTaken + " seconds");
```

What do you think this code will output when executed as part of a `main()` method? Let's take a look:

```
1 2 3 4 5
Time: 25 seconds
```

As you might expect, the results are ordered and predictable because we are using a serial stream. It also took around 25 seconds to process all five results, one at a time. What happens if we replace line 12 with one that uses a `parallelStream()`? The following is some sample output:

```
3 2 1 5 4
Time: 5 seconds
```

As you can see, the results are no longer ordered or predictable. The `map()` and `forEach()` operations on a parallel stream are equivalent to submitting multiple `Runnable` lambda expressions to a pooled thread executor and then waiting for the results.

What about the time required? In this case, our system had enough CPUs for all of the tasks to be run concurrently. If you ran this same code on a computer with fewer processors, it might output 10 seconds, 15 seconds, or some other value. The key is that we've written our code to take advantage of parallel processing when available, so our job is done.

Ordering Results

If your stream operation needs to guarantee ordering and you're not sure if it is serial or parallel, you can replace line 14 with one that uses `forEachOrdered()`:

```
14:     .forEachOrdered(s -> System.out.print(s + " "));
```

This outputs the results in the order in which they are defined in the stream:

```
1 2 3 4 5
Time: 5 seconds
```

While we've lost some of the performance gains of using a parallel stream, our `map()` operation can still take advantage of the parallel stream.

Processing Parallel Reductions

Besides potentially improving performance and modifying the order of operations, using parallel streams can impact how you write your application. A *parallel reduction* is a reduction operation applied to a parallel stream. The results for parallel reductions can differ from what you expect when working with serial streams.

Performing Order-Based Tasks

Since order is not guaranteed with parallel streams, methods such as `findAny()` on parallel streams may result in unexpected behavior. Consider the following example:

```
System.out.print(List.of(1, 2, 3, 4, 5, 6)
   .parallelStream()
   .findAny()
   .get());
```

The JVM allocates a number of threads and returns the value of the first one to return a result, which could be `4`, `2`, and so on. While *neither* the serial

nor the parallel stream is guaranteed to return the first value, the serial stream often does. With a parallel stream, the results are likely to be more random.

What about operations that consider order, such as `findFirst()`, `limit()`, and `skip()`? Order is still preserved, but performance may *suffer* on a parallel stream as a result of a parallel processing task being forced to coordinate all of its threads in a synchronized-like fashion.

Sorting a Parallel Stream

It's possible to sort a parallel stream, although the results might not be what you expect. The following prints the numbers from 1 to 99 in a stochastic, or random, ordering:

```
IntStream.range(1,100).parallel().sorted().forEach(System.out::println);
```

After the call to `sorted()`, the stream is still considered parallel, resulting in the `forEach()` method printing the values in a stochastic ordering. Remember to use an ordered method such as `forEachOrdered()` if you need to guarantee ordering on a parallel stream.

On the plus side, the results of ordered operations on a parallel stream will be consistent with a serial stream. For example, calling `skip(5).limit(2).findFirst()` will return the same result on ordered serial and parallel streams.

Real World Scenario

Creating Unordered Streams

All of the streams you have been working with are considered ordered by default. It is possible to create an unordered stream from an ordered stream, similar to how you create a parallel stream from a serial stream.

```
List.of(1, 2, 3, 4, 5, 6).stream().unordered();
```

This method does not reorder the elements; it just tells the JVM that if an order-based stream operation is applied, the order can be ignored. For example, calling `skip(5)` on an unordered stream will skip any `5` elements, not necessarily the first `5` required on an ordered stream.

For serial streams, using an unordered version has no effect. But on parallel streams, the results can greatly improve performance.

```
List.of(1, 2, 3, 4, 5,
6).stream().unordered().parallel();
```

Even though unordered streams will not be on the exam, if you are developing applications with parallel streams, you should know when to apply an unordered stream to improve performance.

Combining Results with *reduce()*

As you learned in Chapter 10, the stream operation `reduce()` combines a stream into a single object. Recall that the first parameter to the `reduce()` method is called the *identity*, the second parameter is called the *accumulator*, and the third parameter is called the *combiner*. The following is the signature for the method:

```
<U> U reduce(U identity,
   BiFunction<U,? super T,U> accumulator,
   BinaryOperator<U> combiner)
```

We can concatenate a list of `char` values using the `reduce()` method, as shown in the following example:

```
System.out.println(List.of('w', 'o', 'l', 'f')
   .parallelStream()
   .reduce("",
      (s1, c) -> s1 + c,
      (s2, s3) -> s2 + s3));  // wolf
```

The naming of the variables in this stream example is not accidental. We used `c` for `char`, whereas `s1`, `s2`, and `s3` are `String` values.

On parallel streams, the `reduce()` method works by applying the reduction to pairs of elements within the stream to create intermediate values and then combining those intermediate values to produce a final result. Put another way, in a serial stream, `wolf` is built one character at a time. In a parallel stream, the intermediate values `wo` and `lf` are created and then combined.

With parallel streams, we now have to be concerned about order. What if the elements of a string are combined in the wrong order to produce `wlfo` or `flwo`? The Stream API prevents this problem while still allowing streams to be processed in parallel, as long as you follow one simple rule: make sure that the accumulator and combiner produce the same result regardless of the order they are called in.

While this is not in scope for the exam, the accumulator and combiner must be associative, non-interfering, and stateless. Don’t panic; you don’t need to know advanced math terms for the exam!

While the requirements for the input arguments to the `reduce()` method hold true for both serial and parallel streams, you may not have noticed any problems in serial streams because the result was always ordered. With parallel streams, though, order is no longer guaranteed, and any argument that violates these rules is much more likely to produce side effects or unpredictable results.

Let's take a look at an example using a problematic accumulator. In particular, order matters when subtracting numbers; therefore, the following code can output different values depending on whether you use a serial or parallel stream. We can omit a combiner parameter in these examples, as the accumulator can be used when the intermediate data types are the same.

```
System.out.println(List.of(1, 2, 3, 4, 5, 6)
   .parallelStream()
   .reduce(0, (a, b) -> (a - b)));  // PROBLEMATIC ACCUMULATOR
```

It may output `-21`, `3`, or some other value.

You can see other problems if we use an identity parameter that is not truly an identity value. For example, what do you expect the following code to output?

```
System.out.println(List.of("w","o","l","f")
   .parallelStream()
   .reduce("X", String::concat));  // XwXoXlXf
```

On a serial stream, it prints `Xwolf`, but on a parallel stream, the result is `XwXoXlXf`. As part of the parallel process, the identity is applied to multiple elements in the stream, resulting in very unexpected data.

Selecting a *reduce()* Method

Although the one- and two-argument versions of `reduce()` support parallel processing, it is recommended that you use the three-argument version of `reduce()` when working with parallel streams. Providing an explicit combiner method allows the JVM to partition the operations in the stream more efficiently.

Combining Results with *collect()*

Like `reduce()`, the Stream API includes a three-argument version of `collect()` that takes *accumulator* and *combiner* operators along with a *supplier* operator instead of an identity.

```
<R> R collect(Supplier<R> supplier,
   BiConsumer<R, ? super T> accumulator,
   BiConsumer<R, R> combiner)
```

Also, like `reduce()`, the accumulator and combiner operations must be able to process results in any order. In this manner, the three-argument version of `collect()` can be performed as a parallel reduction, as shown in the following example:

```
Stream<String> stream = Stream.of("w", "o", "l",
"f").parallel();
SortedSet<String> set =
stream.collect(ConcurrentSkipListSet::new,
   Set::add,
   Set::addAll);
System.out.println(set);  // [f, l, o, w]
```

Recall that elements in a `ConcurrentSkipListSet` are sorted according to their natural ordering. You should use a concurrent collection to combine the results, ensuring that the results of concurrent threads do not cause a `ConcurrentModificationException`.

Performing parallel reductions with a collector requires additional considerations. For example, if the collection into which you are inserting is an ordered data set, such as a `List`, the elements in the resulting collection must be in the same order, regardless of whether you use a serial or parallel stream. This may reduce performance, though, as some operations cannot be completed in parallel.

Performing a Parallel Reduction on a Collector

While we covered the `Collector` interface in Chapter 10, we didn’t go into detail about its properties. Every `Collector` instance defines a `characteristics()` method that returns a set of `Collector.Characteristics` attributes. When using a `Collector` to perform a parallel reduction, a number of properties must hold true.

Otherwise, the `collect()` operation will execute in a single-threaded fashion.

Requirements for Parallel Reduction with *collect()*

- The stream is parallel.
- The parameter of the `collect()` operation has the `Characteristics.CONCURRENT` characteristic.
- Either the stream is unordered or the collector has the characteristic `Characteristics.UNORDERED`.

For example, while `Collectors.toSet()` does have the `UNORDERED` characteristic, it does not have the `CONCURRENT` characteristic. Therefore, the following is not a parallel reduction even with a parallel stream:

```
parallelStream.collect(Collectors.toSet());  // Not a parallel
reduction
```

The `Collectors` class includes two sets of `static` methods for retrieving collectors, `toConcurrentMap()` and `groupingByConcurrent()`, both of which are `UNORDERED` and `CONCURRENT`. These methods produce `Collector` instances capable of performing parallel reductions efficiently. Like their nonconcurrent counterparts, there are overloaded versions that take additional arguments.

Here is a rewrite of an example from Chapter 10 to use a parallel stream and parallel reduction:

```
Stream<String> ohMy = Stream.of("lions", "tigers",
"bears").parallel();
ConcurrentMap<Integer, String> map = ohMy
   .collect(Collectors.toConcurrentMap(String::length,
      k -> k,
      (s1, s2) -> s1 + "," + s2));
System.out.println(map);             // {5=lions,bears,
6=tigers}
System.out.println(map.getClass());  //
java.util.concurrent.ConcurrentHashMap
```

We use a `ConcurrentMap` reference, although the actual class returned is likely `ConcurrentHashMap`. The particular class is not guaranteed; it will

just be a class that implements the interface `ConcurrentMap`.

Finally, we can rewrite our `groupingBy()` example from Chapter 10 to use a parallel stream and parallel reduction.

```
var ohMy = Stream.of("lions", "tigers", "bears").parallel();
ConcurrentMap<Integer, List<String>> map = ohMy.collect(
   Collectors.groupingByConcurrent(String::length));
System.out.println(map);             // {5=[lions, bears], 6=
[tigers]}
```

As before, the returned object can be assigned to a `ConcurrentMap` reference.

 **Real World Scenario**

Avoiding Stateful Streams

Side effects can appear in parallel streams if your lambda expressions are stateful. A *stateful lambda expression* is one whose result depends on any state that might change during the execution of a pipeline. For example, the following method that filters out even numbers is stateful:

```
public List<Integer> addValues(IntStream source) {
   var data = Collections.synchronizedList(new
ArrayList<Integer>());
   source.filter(s -> s % 2 == 0)
      .forEach(i -> { data.add(i); });  // STATEFUL: DON'T
DO THIS!
   return data;
}
```

Let’s say this method is executed with a serial stream:

```
var list = addValues(IntStream.range(1, 11));
System.out.print(list);                 // [2, 4, 6, 8, 10]
```

Great, the results are in the same order that they were entered. But what if someone else passes in a parallel stream?

```
var list = addValues(IntStream.range(1, 11).parallel());
System.out.print(list);                 // [6, 8, 10, 2, 4]
```

Oh, no: our results no longer match our input order! The problem is that our lambda expression is stateful and modifies a list that is outside our stream. We can fix this solution by rewriting our stream operation to be stateless:

```
public List<Integer> addValuesBetter(IntStream source) {
   return source.filter(s -> s % 2 == 0)
      .boxed()
      .collect(Collectors.toList());
}
```

This method processes the stream and then collects all the results into a new list. It produces the same ordered result on both serial and parallel streams. It is strongly recommended that you avoid stateful operations when using parallel streams to remove any potential data side effects. In fact, they should be avoided in serial streams since doing so limits the code's ability to someday take advantage of parallelization.

Summary

This chapter introduced you to both platform and virtual threads and outlined some of the key concurrency concepts you need to know for the exam (and to be a better software developer!). You should know how to create and define the thread's work using a `Runnable` instance. When working with the Concurrency API, you should also know how to create threads using `Callable` lambda expressions.

At this point, you should know how to concurrently execute tasks using `ExecutorService` like a pro. You should also know which `ExecutorService` instances are available, including scheduled and pooled services.

Thread-safety is about protecting data from being corrupted by multiple threads modifying it at the same time. Java offers many tools to keep data safe, including atomic classes, `synchronized` methods/blocks, the `Lock` framework, and `CyclicBarrier`. The Concurrency API also includes numerous collection classes that handle multithreaded access for you. You should be familiar with the concurrent collections, including the `CopyOnWrite` classes, which create a new underlying structure any time the collection is modified.

When processing tasks concurrently, a variety of potential threading issues can arise. Deadlock, starvation, and livelock can result in programs that appear stuck, while race conditions can result in unpredictable data. For the exam, you need to know only the basic theory behind these concepts. In professional software development, however, finding and resolving such problems is a valuable skill.

Finally, we discussed parallel streams and showed you how to use them to perform parallel decompositions and reductions. Parallel streams can greatly improve the performance of your application. They can also cause unexpected results since the processing is no longer ordered. Remember to avoid stateful lambda expressions, especially when working with parallel streams.

Exam Essentials

Identify the differences between platform threads and virtual threads. Platform threads map to the underlying operating system threads. Virtual threads are lighter weight using a carrier thread only when they need to run. A carrier thread runs on an operating system thread as well. Virtual threads can be run on `Executors.newVirtualThreadPerTaskExecutor()`. Since they are so lightweight, they don't need to be pooled.

Be able to write thread-safe code. Thread-safety is about protecting shared data from concurrent access. A monitor can be used to ensure that only one thread processes a particular section of code at a time. In Java, monitors can be implemented with a `synchronized` block or method or using an instance of `Lock`. `ReentrantLock` has a number of advantages over using a `synchronized` block, including the ability to check whether a lock is available without blocking it, as well as supporting the fair acquisition of locks. To achieve synchronization, two or more threads must coordinate on the same shared object.

Be able to apply the atomic classes. An atomic operation is one that occurs without interference from another thread. The Concurrency API includes a set of atomic classes that are similar to the primitive classes, except that they ensure that operations on them are performed atomically. Know the difference between an atomic variable and one marked with the `volatile` modifier.

Create concurrent tasks with a thread executor service using *Runnable* and *Callable*. An `ExecutorService` creates and manages a single thread or a pool of threads. Instances of `Runnable` and `Callable` can both be submitted to a thread executor and will be completed using the available threads in the service. `Callable` differs from `Runnable` in that `Callable`

returns a generic data type and can throw a checked exception. A `ScheduledExecutorService` can be used to schedule tasks at a fixed rate or with a fixed interval between executions.

Be able to use the concurrent collection classes. The Concurrency API includes numerous collection classes that include built-in support for multithreaded processing, such as `ConcurrentHashMap`. It also includes a class `CopyOnWriteArrayList` that creates a copy of its underlying list structure every time it is modified and is useful in highly concurrent environments.

Identify potential threading problems. Deadlock, starvation, and livelock are three threading problems that can occur and result in threads never completing their task. Deadlock occurs when two or more threads are blocked forever. Starvation occurs when a single thread is perpetually denied access to a shared resource. Livelock is a form of starvation where two or more threads are active but conceptually blocked forever. Finally, race conditions occur when two threads execute at the same time, resulting in an unexpected outcome.

Understand the impact of using parallel streams. The Stream API allows for the easy creation of parallel streams. Using a parallel stream can cause unexpected results, since the order of operations may no longer be predictable. Some operations, such as `reduce()` and `collect()`, require special consideration to achieve optimal performance when applied to a parallel stream.

Review Questions

The answers to the chapter review questions can be found in the Appendix.

1. Given the following code snippet, which options correctly create a parallel stream? (Choose all that apply.)

```
var c = List.of(19, 66);
var s = ThreadLocalRandom.current().doubles();
var p = _________________;
```

   A. `new ParallelStream(s)`