Chapter 18 Concurrency

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- Concurrency
  - Create worker threads using Runnable, Callable and use an ExecutorService to concurrently execute tasks
  - Use java.util.concurrent collections and classes including CyclicBarrier and CopyOnWriteArrayList
  - Write thread-safe code
  - Identify threading problems such as deadlocks and livelocks
- Parallel Streams
  - Develop code that uses parallel streams
  - Implement decomposition and reduction with streams

As you will learn in Chapter 19, "I/O," Chapter 20 "NIO.2," and Chapter 21, "JDBC," computers are capable of reading and writing data to external resources. Unfortunately, as compared to CPU operations, these disk/network operations tend to be extremely slow—so slow, in fact, that 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 or lock up constantly.

Luckily, all modern operating systems support what is known as *multi-threaded processing*. The idea behind multithreaded processing is to allow an application or group of applications to execute multiple tasks at the same time. This allows tasks waiting for other resources to give way to other processing requests.

Since its early days, Java has supported multithreaded programming using the `Thread` class. More recently, the Concurrency API was introduced. It included numerous classes for performing complex thread-based tasks. The idea was simple: managing complex thread interactions is quite difficult for even the most skilled developers; therefore, a set of reusable features was created. The Concurrency API has grown over the years to include numerous classes and frameworks to assist you in developing complex, multithreaded applications. In this chapter, we will introduce you to the concept of threads and provide numerous ways to manage threads using the Concurrency API.

Threads and concurrency tend to be one of the more challenging topics for many programmers to grasp, as problems with threads can be frustrating even for veteran developers to understand. In practice, concurrency issues are among the most difficult problems to diagnose and resolve.

Previous Java certification exams expected you to know details about threads, such as thread life cycles. The 1Z0-816 exam instead covers the basics of threads but focuses more on your knowledge of the Concurrency API. Since we believe that you need to walk before you can run, we provide a basic overview of threads in the first part of this chapter. Be sure you understand the basics of threads both for exam questions and so you better understand the Concurrency API used throughout the rest of the chapter.

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 *multi-threaded process* is one that 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. Refer to Figure 18.1 for an overview of threads and their shared environment within a process.

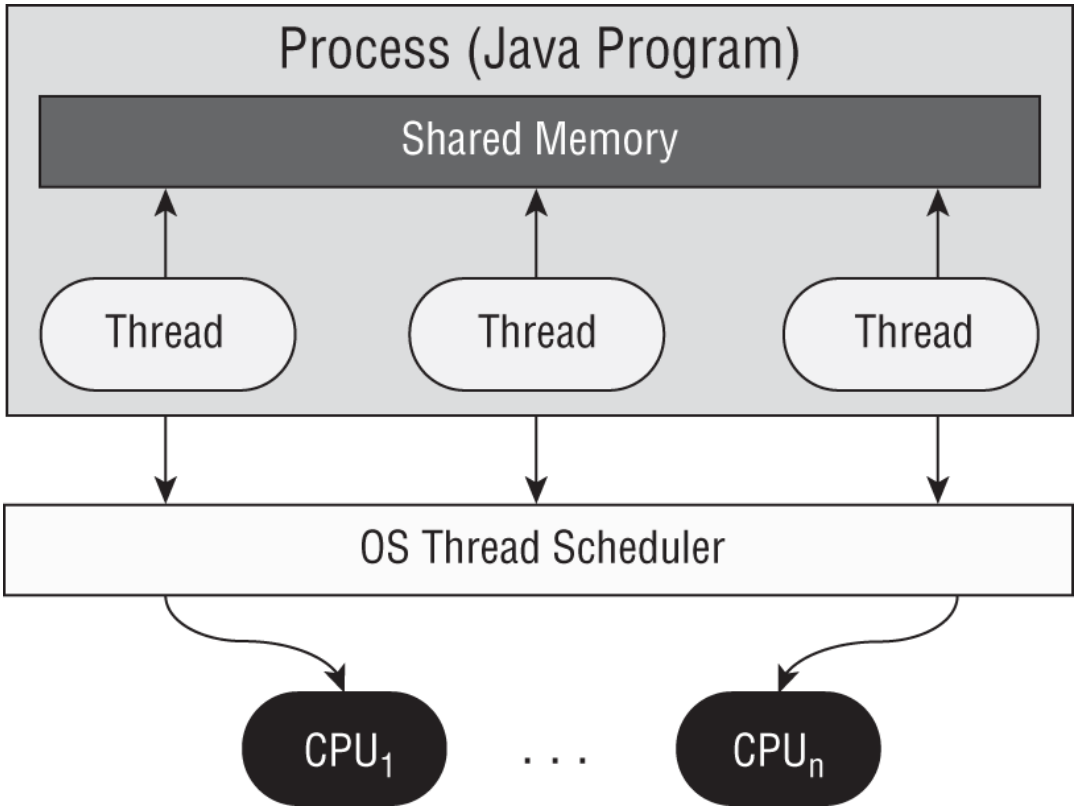

FIGURE 18.1 Process model

Figure 18.1 shows a single process with three threads. It also shows how they are mapped to an arbitrary number of *n* CPUs available within the system. Keep this diagram in mind when we discuss task schedulers later in this section.

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

By shared memory in Figure 18.1, we are generally referring to `static` variables, as well as instance and local variables passed to a thread. Yes, you will finally see how `static` variables can be useful for performing complex, multithreaded tasks! Remember from Chapter 7, "Methods and Encapsulation" that `static` methods and variables are defined on a single class object that all instances share. For example, if one thread updates the value of a `static` object, then this information is immediately available for other threads within the process to read.

Distinguishing Thread Types

It might surprise you that all Java applications, including all of the ones that we have presented in this book, are all multithreaded. Even a simple Java application that prints `Hello World` to the screen is multithreaded. To help you understand this, we introduce the concepts of system threads and user-defined threads.

A *system thread* is created by the JVM and runs in the background of the application. For example, the garbage collection is managed by a system thread that is created by the JVM and runs in the background, helping to free memory that is no longer in use. For the most part, the execution of system-defined threads is invisible to the application developer. When a system-defined thread encounters a problem and cannot recover, such as running out of memory, it generates a Java `Error`, as opposed to an `Exception`.

As discussed in Chapter 16, "Exceptions, Assertions, and Localization," even though it is possible to catch an `Error`, it is considered a poor practice to do so, since it is rare that an application can recover from a system-level failure.

Alternatively, a *user-defined thread* is one created by the application developer to accomplish a specific task. With the exception of parallel streams presented briefly in Chapter 15, "Functional Programming," all of the applications that we have created up to this point have been multithreaded, but they contained only one user-defined thread, which calls the `main( )` method. For simplicity, we commonly refer to threads that contain only a single user-defined thread as a single-threaded application, since we are often uninterested in the system threads.

Although not required knowledge for the exam, 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. Both system and user-defined threads can be marked as daemon threads.

Understanding Thread Concurrency

At the start of the chapter, we mentioned that multithreaded processing allows operating systems to execute threads at the same time. The property of executing multiple threads and processes at the same time is referred to as *concurrency*. Of course, with a single-core CPU system, only one task is actually executing at a given time. Even in multicore or multi-CPU systems, there are often far more threads than CPU processors available. How does the system decide what to execute when there are multiple threads available?

Operating systems use a *thread scheduler* to determine which threads should be currently executing, as shown in Figure 18.1. 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. If there are 10 available

threads, they might each get 100 milliseconds in which to execute, with the process returning to the first thread after the last thread has executed.

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. Be aware that there is often a cost associated with a context switch by way of lost time saving and reloading a thread's state. Intelligent thread schedules 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 than the other thread. A *thread priority* is a numeric value associated with a thread that is taken into consideration by the thread scheduler when determining which threads should currently be executing. In Java, thread priorities are specified as integer values.

Real World Scenario

THE IMPORTANCE OF THREAD SCHEDULING

Even though multicore CPUs are quite common these days, single-core CPUs were the standard in personal computing for many decades. During this time, operating systems developed complex thread-scheduling and context-switching algorithms that allowed users to execute dozens or even hundreds of threads on a single-core CPU system. These scheduling algorithms allowed users to experience the illusion that multiple tasks were being performed at the same time within a single-CPU system. For example, a user could listen to music while writing a paper and receive notifications for new messages.

Since the number of threads requested often far outweighs the number of processors available even in multicore systems, these thread-scheduling algorithms are still employed in operating systems today.

Defining a Task with *Runnable*

As we mentioned in Chapter 15, `java.lang.Runnable` is a functional interface that takes no arguments and returns no data. The following is the definition of the `Runnable` interface:

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

The `Runnable` interface is commonly used to define the task or work a thread will execute, separate from the main application thread. We will be relying on the `Runnable` interface throughout this chapter, especially when we discuss applying parallel operations to streams.

The following lambda expressions each implement the `Runnable` interface:

```
Runnable sloth = () -> System.out.println("Hello World");
Runnable snake = () -> {int i=10; i++;};
Runnable beaver = () -> {return;};
Runnable coyote = () -> {};
```

Notice that all of these lambda expressions start with a set of empty parentheses, `()`. Also, none of the lambda expressions returns a value. The following lambdas, while valid for other functional interfaces, are not compatible with `Runnable` because they return a value.

```
Runnable capybara = () -> "";                 // DOES NOT COMPILE
Runnable Hippopotamus = () -> 5;              // DOES NOT COMPILE
Runnable emu = () -> {return new Object();};  // DOES NOT COMPILE
```

CREATING *RUNNABLE* CLASSES

Even though `Runnable` is a functional interface, many classes implement it directly, as shown in the following code:

```
public class CalculateAverage implements Runnable {
   public void run() {
      // Define work here
   }
}
```

It is also useful if you need to pass information to your `Runnable` object to be used by the `run()` method, such as in the following constructor:

```
public class CalculateAverages implements Runnable {
   private double[] scores;
public CalculateAverages(double[] scores) {
      this.scores = scores;
   }
   public void run() {
      // Define work here that uses the scores object
   }
}
```

In this chapter, we focus on creating lambda expressions that implicitly implement the `Runnable` interface. Just be aware that it is commonly used in class definitions.

Creating a Thread

The simplest way to execute a thread is by using the `java.lang.Thread` class. Executing a task with `Thread` is a two-step process. First, you define the `Thread` with the corresponding task to be done. Then, you start the task by using the `Thread.start()` method.

As we will discuss later in the chapter, Java does not provide any guarantees about the order in which a thread will be processed once it is started. It may be executed immediately or delayed for a significant amount of time.

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.

Defining the task that a `Thread` instance will execute can be done two ways in Java:

- Provide a `Runnable` object or lambda expression to the `Thread` constructor.
- Create a class that extends `Thread` and overrides the `run()` method.

The following are examples of these techniques:

```
public class PrintData implements Runnable {
   @Override public void run() { // Overrides method in Runnable
      for(int i = 0; i < 3; i++)
         System.out.println("Printing record: "+i);
   }
   public static void main(String[] args) {
      (new Thread(new PrintData())).start();
   }
}

public class ReadInventoryThread extends Thread {
   @Override public void run() { // Overrides method in Thread
      System.out.println("Printing zoo inventory");
   }
   public static void main(String[] args) {
      (new ReadInventoryThread()).start();
```

```
    }
}
```

The first example creates a `Thread` using a `Runnable` instance, while the second example uses the less common practice of extending the `Thread` class and overriding the `run()` method. Anytime you create a `Thread` instance, make sure that you remember to start the task with the `Thread.start()` method. This starts the task in a separate operating system thread.

Let's try this. What is the output of the following code snippet using these two classes?

```
2: public static void main(String[] args) {
3:    System.out.println("begin");
4:    (new ReadInventoryThread()).start();
5:    (new Thread(new PrintData())).start();
6:    (new ReadInventoryThread()).start();
7:    System.out.println("end");
8: }
```

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

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

This sample uses a total of four threads—the `main()` user thread and three additional threads created on lines 4–6. 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. For example,

lines 5 and 6 may be executed before the thread created on line 4 finishes. The opposite of this behavior is a *synchronous* task in which the program waits (or blocks) on line 4 for the thread to finish executing before moving on to the next line. The vast majority of method calls used in this book have been synchronous up until now.

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

CALLING *RUN()* INSTEAD OF *START()*

Be careful with code that attempts to start a thread by calling `run()` instead of `start()`. Calling `run()` on a `Thread` or a `Runnable` does not actually start a new thread. While the following code snippets will compile, none will actually execute a task on a separate thread:

```
System.out.println("begin");
(new ReadInventoryThread()).run();
(new Thread(new PrintData())).run();
(new ReadInventoryThread()).run();
System.out.println("end");
```

Unlike the previous example, each line of this code will wait until the `run()` method is complete before moving on to the next line. Also unlike the previous program, the output for this code sample will be the same each time it is executed.

In general, you should extend the `Thread` class only under specific circumstances, such as when you are creating your own priority-based thread. In most situations, you should implement the `Runnable` interface rather than extend the `Thread` class.

We conclude our discussion of the `Thread` class here. While previous versions of the exam were quite focused on understanding the difference be-

tween extending `Thread` and implementing `Runnable`, the exam now strongly encourages developers to use the Concurrency API.

For the exam, you also do not need to know about other thread-related methods, such as `Object.wait()`, `Object.notify()`, `Thread.join()`, etc. In fact, you should avoid them in general and use the Concurrency API as much as possible. It takes a large amount of skill (and some luck!) to use these methods correctly.

Real World Scenario

FOR INTERVIEWS, BE FAMILIAR WITH THREAD-CREATION OPTIONS

Despite that the exam no longer focuses on creating threads by extending the `Thread` class and implementing the `Runnable` interface, it is extremely common when interviewing for a Java development position to be asked to explain the difference between extending the `Thread` class and implementing `Runnable`.

If asked this question, you should answer it accurately. You should also mention that you can now create and manage threads indirectly using an `ExecutorService`, which we will discuss in the next section.

Polling with Sleep

Even though multithreaded programming allows you to execute multiple tasks at the same time, one thread often needs to wait for the results of another thread to proceed. One solution is to use polling. *Polling* is the process of intermittently checking data at some fixed interval. For example, let's say you have a thread that modifies a shared `static counter` value and your `main()` thread is waiting for the thread to increase the value to greater than `100`, as shown in the following class:

```
public class CheckResults {
   private static int counter = 0;
   public static void main(String[] args) {
```

```
        new Thread(() -> {
           for(int i = 0; i < 500; i++) CheckResults.counter++;
        }).start();
        while(CheckResults.counter < 100) {
           System.out.println("Not reached yet");
        }
        System.out.println("Reached!");
     }
  }
```

How many times does this program print `Not reached yet`? The answer is, we don't know! It could output zero, ten, or a million times. If our thread scheduler is particularly poor, it could operate infinitely! Using a `while()` loop to check for data without some kind of delay is considered a bad coding practice as it ties up CPU resources for no reason.

We can improve this result by using the `Thread.sleep()` method to implement polling. The `Thread.sleep()` method requests the current thread of execution rest for a specified number of milliseconds. When used inside the body of the `main()` method, the thread associated with the `main()` method will pause, while the separate thread will continue to run. Compare the previous implementation with the following one that uses `Thread.sleep()`:

```
  public class CheckResults {
     private static int counter = 0;
     public static void main(String[] a) throws InterruptedException {
        new Thread(() -> {
           for(int i = 0; i < 500; i++) CheckResults.counter++;
        }).start();
        while(CheckResults.counter < 100) {
           System.out.println("Not reached yet");
           Thread.sleep(1000); // 1 SECOND
        }
        System.out.println("Reached!");
     }
  }
```

In this example, we delay 1,000 milliseconds at the end of the loop, or 1 second. While this may seem like a small amount, we have now prevented a possibly infinite loop from executing and locking up our program. Notice that we also changed the signature of the `main()` method, since `Thread.sleep()` throws the checked `InterruptedException`. Alternatively, we could have wrapped each call to the `Thread.sleep()` method in a `try/catch` block.

How many times does the `while()` loop execute in this revised class? Still unknown! While polling does prevent the CPU from being overwhelmed with a potentially infinite loop, it does not guarantee when the loop will terminate. For example, the separate thread could be losing CPU time to a higher-priority process, resulting in multiple executions of the `while()` loop before it finishes.

Another issue to be concerned about is the shared `counter` variable. What if one thread is reading the `counter` variable while another thread is writing it? The thread reading the shared variable may end up with an invalid or incorrect value. We will discuss these issues in detail in the upcoming section on writing thread-safe code.

Creating Threads with the Concurrency API

Java includes 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 for you.

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 anytime 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. As you may remember from Chapter 16, the factory pattern is a creational pattern in which the underlying implementation details of the object creation are hidden from us. You will see the factory pattern used again throughout Chapter 20.

Let's start with a simple example using the `newSingleThreadExecutor()` method to obtain an `ExecutorService` instance and the `execute()` method to perform asynchronous tasks.

```
import java.util.concurrent.*;
public class ZooInfo {
   public static void main(String[] args) {
      ExecutorService service = null;
      Runnable task1 = () ->
         System.out.println("Printing zoo inventory");
      Runnable task2 = () -> {for(int i = 0; i < 3; i++)
            System.out.println("Printing record: "+i);};
      try {
         service = Executors.newSingleThreadExecutor();
         System.out.println("begin");
         service.execute(task1);
         service.execute(task2);
         service.execute(task1);
         System.out.println("end");
      } finally {
         if(service != null) service.shutdown();
      }
   }
}
```

As you may notice, this is just a rewrite of our earlier `PrintData` and `ReadInventoryThread` classes to use lambda expressions and an `ExecutorService` instance.

In this example, we use the `Executors.newSingleThreadExecutor()` method to create the service. Unlike our earlier example, in which we

had three extra threads for newly created tasks, this example uses only one, which means that the threads will order their results. For example, the following is a possible output for this code snippet:

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

With a single-thread executor, results 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`.

Shutting Down a Thread Executor

Once you have finished using a thread executor, it is important that you call the `shutdown()` method. A thread executor creates a non-daemon thread on the first task that is executed, so failing to call `shutdown()` will result in your application never terminating.

The shutdown process for a thread executor involves first rejecting any new tasks submitted to the thread executor while continuing to execute any previously submitted tasks. During this time, calling `isShutdown()` will return `true`, while `isTerminated()` will return `false`. If a new task is submitted to the thread executor while it is shutting down, a `RejectedExecutionException` will be thrown. Once all active tasks have been completed, `isShutdown()` and `isTerminated()` will both return `true`. Figure 18.2 shows the life cycle of an `ExecutorService` object.

FIGURE 18.2 `ExecutorService` life cycle

For the exam, you should be aware that `shutdown()` does not actually stop any tasks that have already been submitted to the thread executor.

What if you want to cancel all running and upcoming tasks? The `ExecutorService` provides a method called `shutdownNow()`, which *attempts to stop* all running tasks and discards any that have not been started yet. It is possible to create a thread that will never terminate, so any attempt to interrupt it may be ignored. Lastly, `shutdownNow()` returns a `List<Runnable>` of tasks that were submitted to the thread executor but that were never started.

As you learned in Chapter 16, resources such as thread executors should be properly closed to prevent memory leaks. Unfortunately, the `ExecutorService` interface does not extend the `AutoCloseable` interface, so you cannot use a try-with-resources statement. You can still use a `finally` block, as we do throughout this chapter. While not required, it is considered a good practice to do so.

Submitting Tasks

You can submit tasks to an `ExecutorService` instance multiple ways. The first method we presented, `execute()`, is inherited from the `Executor` interface, which the `ExecutorService` interface extends. The `execute()`

method takes a `Runnable` lambda expression or 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 18.1 shows the five methods, including `execute()` and two `submit()` methods, which you should know for the exam. Don't worry if you haven't seen `Future` or `Callable` before; we will discuss them in detail shortly.

In practice, using the `submit()` method is quite similar to using the `execute()` method, except that the `submit()` method returns a `Future` instance that can be used to determine whether the task has completed execution.

TABLE 18.1 `ExecutorService` methods

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

---

SUBMITTING TASKS: *EXECUTE()* VS. *SUBMIT()*

As you might have noticed, the `execute()` and `submit()` methods are nearly identical when applied to `Runnable` expressions. The `submit()` method has the obvious advantage of doing the same thing `execute()` does, but with a return object that can be used to track the result. Because of this advantage and the fact that `execute()` does not support `Callable` expressions, we tend to prefer `submit()` over `execute()`, even if you don't store the `Future` reference. Therefore, we use `submit()` in the majority of the examples in this chapter.

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.

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 `java.util.concurrent.Future<V>` 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 18.2 includes useful methods for determining the state of a task.

TABLE 18.2 `Future` methods

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

The following is an updated version of our earlier polling example `CheckResults` class, which 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 {
      ExecutorService service = null;
      try {
         service = Executors.newSingleThreadExecutor();
         Future<?> result = service.submit(() -> {
            for(int i = 0; i < 500; i++) CheckResults.counter++;
```

```
      });
      result.get(10, TimeUnit.SECONDS);
      System.out.println("Reached!");
   } catch (TimeoutException e) {
      System.out.println("Not reached in time");
   } finally {
      if(service != null) service.shutdown();
   } } }
```

This example is similar to our earlier polling implementation, but it 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. It 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`. We present the full list of `TimeUnit` values in Table 18.3 in increasing order of duration. Numerous methods in the Concurrency API use the `TimeUnit` enum.

TABLE 18.3 `TimeUnit` values

Enum name	Description
`TimeUnit.NANOSECONDS`	Time in one-billionth of a second (1/1,000,000,000)
`TimeUnit.MICROSECONDS`	Time in one-millionth of a second (1/1,000,000)
`TimeUnit.MILLISECONDS`	Time in one-thousandth 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

Introducing *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;
}
```

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 throughout this chapter, 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.

Unlike `Runnable`, in which the `get()` methods always return `null`, the `get()` methods on a `Future` instance return the matching generic type (which could also be a `null` value).

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

```
import java.util.concurrent.*;
public class AddData {
   public static void main(String[] args) throws Exception {
      ExecutorService service = null;
      try {
         service = Executors.newSingleThreadExecutor();
         Future<Integer> result = service.submit(() -> 30 + 11);
         System.out.println(result.get());   // 41
      } finally {
         if(service != null) service.shutdown();
      }
   }
}
```

The results could have also been obtained using `Runnable` and some shared, possibly `static`, object, although this solution that relies on `Callable` is a lot simpler and easier to follow.

Waiting for All Tasks to Finish

After submitting a set of tasks to a thread executor, it is common to wait for the results. As you saw in the previous sections, one solution is to call `get()` on each `Future` object returned by the `submit()` method. If we don't need the results of the tasks and are finished using our thread executor, there is a simpler approach.

First, we shut down the thread executor using the `shutdown()` method. Next, we use the `awaitTermination()` method available for all thread ex-

ecutors. The method waits the specified time to complete all tasks, returning sooner if all tasks finish or an `InterruptedException` is detected. You can see an example of this in the following code snippet:

```
ExecutorService service = null;
try {
   service = Executors.newSingleThreadExecutor();
   // Add tasks to the thread executor
   …
} finally {
   if(service != null) service.shutdown();
}
if(service != null) {
   service.awaitTermination(1, TimeUnit.MINUTES);

   // Check whether all tasks are finished
   if(service.isTerminated()) System.out.println("Finished!");
   else System.out.println("At least one task is still running");
}
```

In this example, we submit a number of tasks to the thread executor and then shut down the thread executor and wait up to one minute for the results. Notice that we can call `isTerminated()` after the `awaitTermination()` method finishes to confirm that all tasks are actually finished.

If `awaitTermination()` is called before `shutdown()` within the same thread, then that thread will wait the full timeout value sent with `awaitTermination()`.

Submitting Task Collections

The last two methods listed in Table 18.2 that you should know for the exam are `invokeAll()` and `invokeAny()`. Both of these methods execute synchronously and take a `Collection` of tasks. Remember that by syn-

chronous, we mean that unlike the other methods used to submit tasks to a thread executor, these methods will wait until the results are available before returning control to the enclosing program.

The `invokeAll()` method executes all tasks in a provided collection and returns a `List` of ordered `Future` instances, with one `Future` instance corresponding to each submitted task, in the order they were in the original collection.

```
20: ExecutorService service = …
21: System.out.println("begin");
22: Callable<String> task = () -> "result";
23: List<Future<String>>  list = service.invokeAll(
24:    List.of(task, task, task));
25: for (Future<String> future : list) {
26:    System.out.println(future.get());
27: }
28: System.out.println("end");
```

In this example, the JVM waits on line 23 for all tasks to finish before moving on to line 25. Unlike our earlier examples, this means that `end` will always be printed last. Also, even though `future.isDone()` returns `true` for each element in the returned `List`, a task could have completed normally or thrown an exception.

On the other hand, the `invokeAny()` method executes a collection of tasks and returns the result of one of the tasks that successfully completes execution, cancelling all unfinished tasks. While the first task to finish is often returned, this behavior is not guaranteed, as any completed task can be returned by this method.

```
20: ExecutorService service = …
21: System.out.println("begin");
22: Callable<String> task = () -> "result";
23: String data = service.invokeAny(List.of(task, task, task));
24: System.out.println(data);
25: System.out.println("end");
```

As before, the JVM waits on line 23 for a completed task before moving on to the next line. The other tasks that did not complete are cancelled.

For the exam, remember that the `invokeAll()` method will wait indefinitely until all tasks are complete, while the `invokeAny()` method will wait indefinitely until at least one task completes. The `ExecutorService` interface also includes overloaded versions of `invokeAll()` and `invokeAny()` that take a `timeout` value and `TimeUnit` parameter.

Scheduling Tasks

Oftentimes 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. The `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();
```

We could store an instance of `ScheduledExecutorService` in an `ExecutorService` variable, although doing so would mean we'd have to cast the object to call any scheduled methods.

Refer to Table 18.4 for our summary of `ScheduledExecutorService` methods.

TABLE 18.4 `ScheduledExecutorService` methods

Method Name	Description
`schedule(Callable<V> callable, long delay, TimeUnit unit)`	Creates and executes a `Callable` task after the given delay
`schedule(Runnable command, long delay, TimeUnit unit)`	Creates and executes a `Runnable` task after the given delay
`scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)`	Creates and executes a `Runnable` task after the given initial delay, creating a new task every period value that passes
`scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)`	Creates and executes a `Runnable` task after the given initial delay and subsequently with the given delay between the termination of one execution and the commencement of the 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 18.4 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:

```
ScheduledExecutorService 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 task, at which point they will just wait in the queue. Also, if the `ScheduledExecutorService` is 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 processing a task and send out notifications if it is not finished or even call `schedule()` again to delay processing.

The last two methods in Table 18.4 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 completing some 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 fin-

ished. The following example executes a `Runnable` task every minute, following an initial five-minute delay:

```
service.scheduleAtFixedRate(command, 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 your boss came by your desk every minute and dropped off a piece of paper. Now imagine 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, then the next task will start at 12:07.

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

The `scheduleWithFixedDelay()` 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, he doesn't need to check at some specific time, just after enough time has passed for it to become low on stock again.

If you are familiar with creating Cron jobs in Linux to schedule tasks, then you should know that `scheduleAtFixedRate()` is the closest built-in Java equivalent.

Increasing Concurrency with Pools

All of our examples up until now have been with single-thread executors, 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!

We now present three additional factory methods in the `Executors` class that act on a pool of threads, rather than on a single thread. A *thread pool* is a group of pre-instantiated reusable threads that are available to perform a set of arbitrary tasks. Table 18.5 includes our two previous single-thread executor methods, along with the new ones that you should know for the exam.

TABLE 18.5 `Executors` factory methods

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

As shown in Table 18.5, 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 a 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.

The `newFixedThreadPool()` takes a number of threads and allocates them all upon creation. As long as our number of tasks is less than our number of threads, all tasks will be executed concurrently. If at any point the number of tasks exceeds the number of threads in the pool, they will wait in a similar manner as you saw with a single-thread executor. In fact, calling `newFixedThreadPool()` with a value of `1` is equivalent to calling `newSingleThreadExecutor()`.

The `newCachedThreadPool()` method will create a thread pool of unbounded size, allocating a new thread anytime one is required or all existing threads are busy. This is commonly used for pools that require executing many short-lived asynchronous tasks. For long-lived processes, usage of this executor is strongly discouraged, as it could grow to encompass a large number of threads over the application life cycle.

The `newScheduledThreadPool()` is identical to the `newFixedThreadPool()` method, except that it returns an instance of `ScheduledExecutorService` and is therefore compatible with scheduling tasks.

Real World Scenario

CHOOSING A POOL SIZE

In practice, choosing an appropriate pool size requires some thought. In general, you want at least a handful more threads than you think you will ever possibly need. On the other hand, you don't want to choose so many threads that your application uses up too many resources or too much CPU processing power. Oftentimes, the number of CPUs available is used to determine the thread pool size using this command:

```
Runtime.getRuntime().availableProcessors()
```

It is a common practice to allocate threads based on the number of CPUs.

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. We present the following code to repre-

sent this conceptually, choosing a thread pool size so that all tasks can be run concurrently:

```
import java.util.concurrent.*;
public class SheepManager {
   private int sheepCount = 0;
   private void incrementAndReport() {
      System.out.print((++sheepCount)+" ");
   }
   public static void main(String[] args) {
      ExecutorService service = null;
      try {
         service = Executors.newFixedThreadPool(20);
         SheepManager manager = new SheepManager();
         for(int i = 0; i < 10; i++)
            service.submit(() -> manager.incrementAndReport());
      } finally {
         if(service != null) service.shutdown();
      }
   }
}
```

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 all 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 execute the right side of the expression, reading 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 18.3 demon-

strates this problem with two threads, assuming that `sheepCount` has a starting value of `1`.

You can see in Figure 18.3 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 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.

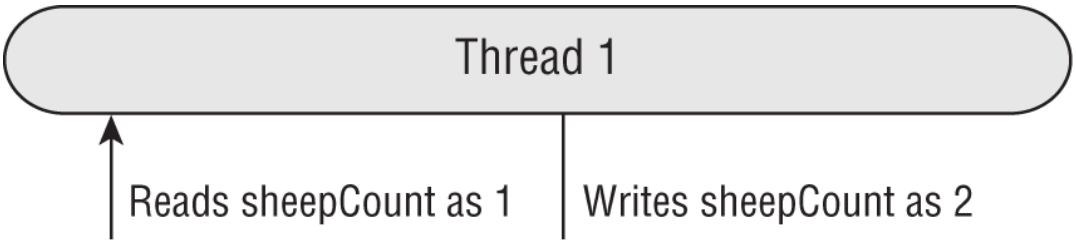

FIGURE 18.3 Lack of thread synchronization

Protecting Data with Atomic Classes

One way to improve our sheep counting example is to use the `java.util.concurrent.atomic` package. As with many of the classes in the Concurrency API, these classes exist to make your life easier.

In our first `SheepManager` sample output, the same values were printed twice, with the highest counter being `9` instead of `10`. As we demonstrated in the previous section, the increment operator `++` is not thread-safe. Furthermore, the reason that it is not thread-safe is that 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 by another thread. A thread-safe atomic version of the increment operator would be one that performed the read and write of the variable as a single operation, not allowing any other threads to access the variable during the operation. Figure 18.4 shows the result of making the `sheepCount` variable atomic.

Figure 18.4 resembles our earlier Figure 18.3, except that reading and writing the data is atomic with regard to the `sheepCount` variable. 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 result in order.

Since accessing primitives and references in Java is common in shared environments, the Concurrency API includes numerous useful classes that are conceptually the same as our primitive classes but that support atomic operations. Table 18.6 lists the atomic classes with which you should be familiar for the exam.

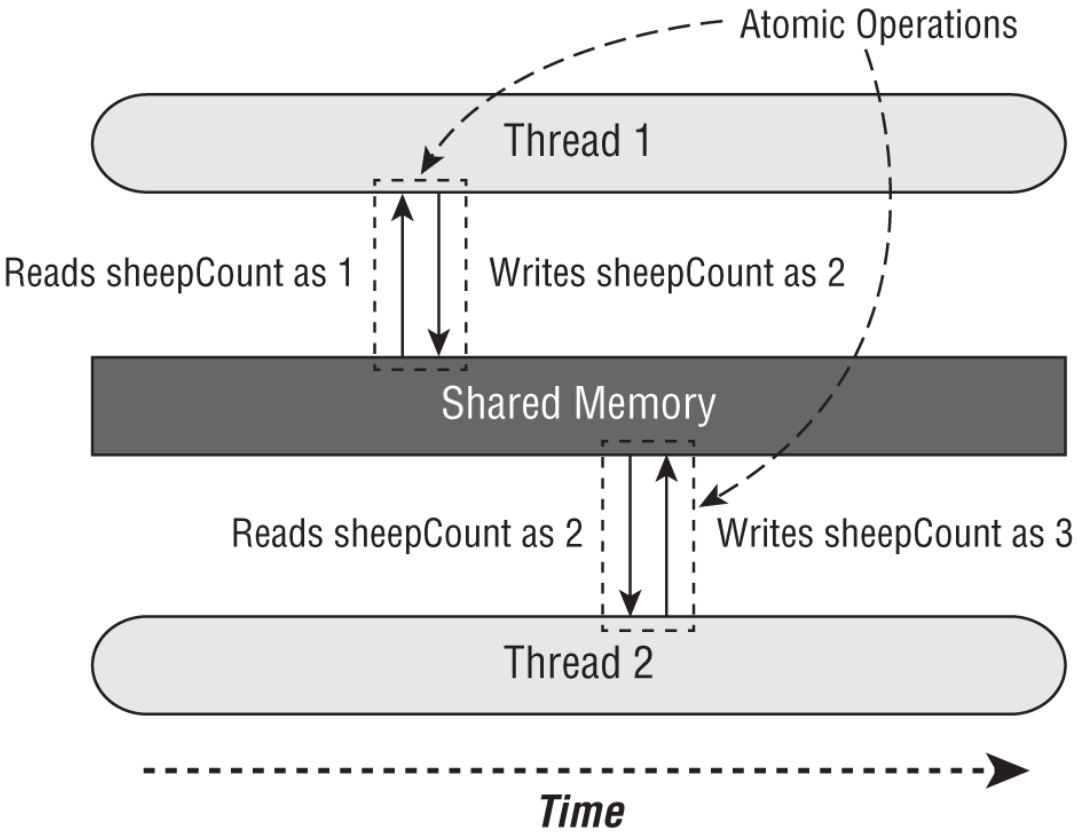

FIGURE 18.4 Thread synchronization using atomic operations

TABLE 18.6 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 primitive 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 18.7.

In the following example, we update our `SheepManager` class with an `AtomicInteger`:

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

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'll 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.

TABLE 18.7 Common atomic methods

Method name	Description
`get()`	Retrieves the current value
`set()`	Sets the given value, equivalent to the assignment `=` operator
`getAndSet()`	Atomically sets the new value and returns the 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--`

Improving Access with Synchronized Blocks

While atomic classes are great at protecting single variables, they aren't particularly useful if you need to execute a series of commands or call a method. 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, also called a *lock*, to synchronize access. A *monitor* 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:

```
SheepManager 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 in the block. In this manner, a thread "acquires the lock" for the monitor. If the lock is available, a single thread will enter the block, acquiring the lock and preventing all other threads from entering. While the first thread is executing the block, all threads that arrive will attempt to acquire the same lock and wait for the first thread to finish. Once a 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`. For example, synchronizing on different objects would not actually order the results.

Let's revisit our `SheepManager` example 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:

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

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, each thread would be created one at a

time, but they may all still execute and perform their work at the same time, resulting in the same type of output that you saw earlier. Diagnosing and resolving threading problems is often one of the most difficult tasks in any programming language.

We now present a corrected version of the `SheepManager` class, which does order the workers.

```
import java.util.concurrent.*;
public class SheepManager {
   private int sheepCount = 0;
   private void incrementAndReport() {
      synchronized(this) {
         System.out.print((++sheepCount)+" ");
      }
   }
   public static void main(String[] args) {
      ExecutorService service = null;
      try {
         service = Executors.newFixedThreadPool(20);
         var manager = new SheepManager();
         for(int i = 0; i < 10; i++)
            service.submit(() -> manager.incrementAndReport());
      } finally {
         if(service != null) service.shutdown();
      }
   }
}
```

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, so long as it was the same object. For example, the following code snippet would have also worked:

```
private final Object herd = new Object();
private void incrementAndReport() {
   synchronized(herd) {
      System.out.print((++sheepCount)+" ");
   }
}
```

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

We could have used an atomic variable along with the `synchronized` block in this example, although it is unnecessary. Since `synchronized` blocks allow only one thread to enter, we're not gaining any improvement by using an atomic variable if the only time that we access the variable is within a `synchronized` block.

Synchronizing on Methods

In the previous example, we established our monitor using `synchronized(this)` around the body of the method. Java actually provides a more convenient compiler enhancement 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:

```
private void incrementAndReport() {
   synchronized(this) {
```

```
            System.out.print((++sheepCount)+" ");
        }
    }
    private synchronized void incrementAndReport() {
        System.out.print((++sheepCount)+" ");
    }
```

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:

```
    public static void printDaysWork() {
        synchronized(SheepManager.class) {
            System.out.print("Finished work");
        }
    }
    public static synchronized void printDaysWork() {
        System.out.print("Finished work");
    }
```

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.

AVOID SYNCHRONIZATION WHENEVER POSSIBLE

Correctly using the `synchronized` keyword can be quite challenging, especially if the data you are trying to protect is available to dozens of methods. Even when the data is protected, though, the performance cost for using it can be high.

In this chapter, we present many classes within the Concurrency API that are a lot easier to use and more performant than synchronization. Some you have seen already, like the atomic classes, and others we'll be covering shortly, including the `Lock` framework, concurrent collections, and cyclic barriers.

While you may not be familiar with all of the classes in the Concurrency API, you should study them carefully if you are writing a lot of multithreaded applications. They contain a wealth of methods that manage complex processes for you in a thread-safe and performant manner.

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 hang forever.

The Concurrency API includes the `Lock` interface that is conceptually similar to using the `synchronized` keyword, but with a lot more bells and whistles. Instead of synchronizing on any `Object` , though, we can "lock" only on an object that implements the `Lock` interface.

Applying a *ReentrantLock* Interface

Using the `Lock` interface is pretty easy. 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. As we'll see in the next section, the `Lock` solution has a number of features not available to the `synchronized` block.

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

// Protected code
}

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

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

While certainly not required, it is a good practice to use a `try` / `finally` block with `Lock` instances. This ensures any acquired locks are properly released.

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.

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()`. As far as which thread gets the lock next, that depends on the parameters used to create the `Lock` object.

The `ReentrantLock` class contains a constructor that can be used to send a `boolean` "fairness" parameter. If set to `true`, then the lock will usually be granted to each thread in the order it was requested. It is `false` by default when using the no-argument constructor. 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 make sure that you only release a lock that you actually have. If you attempt to release a lock that you do not have, you will get an exception at runtime.

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

The `Lock` interface includes four methods that you should know for the exam, as listed in Table 18.8.

TABLE 18.8 `Lock` methods

Method	Description
`void lock()`	Requests a lock and blocks until lock is acquired
`void unlock()`	Releases a lock
`boolean tryLock()`	Requests a lock and returns immediately. Returns a `boolean` indicating whether the lock was successfully acquired
`boolean tryLock(long,TimeUnit)`	Requests a lock and blocks up to the specified time until lock is required. Returns a `boolean` indicating whether the 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 18.8 includes two additional methods that make the `Lock` interface a lot safer to use than a `synchronized` block.

For convenience, we'll be using the following `printMessage()` method for the code in this section:

```
public static void printMessage(Lock lock) {
   try {
      lock.lock();
   } finally {
      lock.unlock();
   }
}
```

tryLock()

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 or not a lock is available.

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

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

When you run this code, it could produce either message, depending on the order of execution. 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.

It is imperative that your program always checks the return value of the `tryLock()` method. It tells your program whether the lock needs to be released later.

tryLock(long,TimeUnit)

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, then it will immediately return with it. If a lock is unavailable, though, it will wait up to the specified time limit for the lock.

The following code snippet uses the overloaded version of `tryLock(long,TimeUnit)`:

```
Lock lock = new ReentrantLock();
new Thread(() -> printMessage(lock)).start();
if(lock.tryLock(10,TimeUnit.SECONDS)) {
   try {
      System.out.println("Lock obtained, entering protected code");
   } finally {
      lock.unlock();
   }
} else {
   System.out.println("Unable to acquire lock, doing something else");
}
```

The code is the same as before, except this time one of the threads waits up to 10 seconds to acquire the lock.

Duplicate Lock Requests

The `ReentrantLock` class maintains a counter of the number of times a lock has been given 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?

```
Lock lock = new ReentrantLock();
if(lock.tryLock()) {
   try {
```

```
            lock.lock();
            System.out.println("Lock obtained, entering protected code");
        } finally {
            lock.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`:

```
new Thread(() -> System.out.print(lock.tryLock())).start();
```

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 it was requested.

The Concurrency API includes other lock-based classes, although `ReentrantLock` is the only one you need to know for the exam.

While not on the exam, `ReentrantReadWriteLock` is a really useful class. It includes separate locks for reading and writing data and is useful on data structures where reads are far more common than writes. For example, if you have a thousand threads reading data but only one thread writing data, this class can help you maximize concurrent access.

Orchestrating Tasks with a *CyclicBarrier*

We started thread-safety discussing protecting individual variables and then moved on to blocks of code and locks. We complete our discussion of thread-safety by discussing how to orchestrate complex tasks across many things.

Our zoo workers are back, and this time they are cleaning pens. Imagine that there is a lion pen that needs to be emptied, cleaned, and then filled back up 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. For now, let's start with a code sample without a `CyclicBarrier`.

```
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) {
       ExecutorService service = null;
       try {
          service = Executors.newFixedThreadPool(4);
          var manager = new LionPenManager();
          for (int i = 0; i < 4; i++)
             service.submit(() -> manager.performTask());
       } finally {
          if (service != null) service.shutdown();
       }
    }
}
```

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 within a single thread the results are ordered, among multiple workers the output is entirely random. 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. In our conceptual example, this would be quite chaotic and would not lead to a very clean cage.

We can improve these results by using the `CyclicBarrier` class. The `CyclicBarrier` 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.

The following is a reimplementation of our `LionPenManager` class that uses `CyclicBarrier` objects to coordinate access:

```
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) {
      ExecutorService service = null;
      try {
         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));
      } finally {
         if (service != null) service.shutdown();
      }
   }
}
```

In this example, we have updated `performTask()` to use `CyclicBarrier` objects. Like synchronizing on the same object, coordinating a task with a `CyclicBarrier` requires the object to be `static` or passed to the thread performing the task. We also add a `try`/`catch` block in the `performTask()` method, as the `await()` method throws multiple checked exceptions.

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 all 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 called a `Runnable` method upon completion.

THREAD POOL SIZE AND CYCLIC BARRIER LIMIT

If you are using a thread pool, make sure that you set the number of available threads to be at least as large as your `CyclicBarrier` limit value. For example, what if we changed the code in the previous example to allocate only two threads, such as in the following snippet?

```
ExecutorService service = Executors.newFixedThreadPool(2);
```

In this case, the code will hang indefinitely. The barrier would never be reached as the only threads available in the pool are stuck waiting for the barrier to be complete. This would result in a deadlock, which will be discussed shortly.

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.

There is a slight loss in performance to be expected from using a `CyclicBarrier`. For example, one worker may be incredibly slow at removing lions, resulting in the other three workers waiting for him to finish. Since we can't start cleaning the pen while it is full of lions, though, this solution is about as concurrent as we can make it.

REUSING *CYCLICBARRIER*

After a `CyclicBarrier` 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()`, then 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 14, "Generics and Collections." In this section, we will 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:

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

```
var foodData = new ConcurrentHashMap<String, Integer>();
foodData.put("penguin", 1);
```

```
foodData.put("flamingo", 2);
for(String key: foodData.keySet())
   foodData.remove(key);
```

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 anytime that you are going to have multiple threads modify a collections object outside a `synchronized` block or method, even if you don't expect a concurrency problem. On the other hand, immutable or read-only objects can be accessed by any number of threads without a concurrent collection.

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.

In the same way that we instantiate an `ArrayList` object but pass around a `List` reference, it is considered a good practice to instantiate a concurrent collection but pass it around using a nonconcurrent interface whenever possible. In some cases, the callers may need to know that it is a concurrent collection so that a concurrent interface or class is appropri-

ate, but for the majority of circumstances, that distinction is not necessary.

Table 18.9 lists the common concurrent classes with which you should be familiar for the exam.

TABLE 18.9 Concurrent collection classes

Class name	Java Collections Framework interfaces	Elements ordered?	Sorted?	Blocking?
`ConcurrentHashMap`	`ConcurrentMap`	No	No	No
`ConcurrentLinkedQueue`	`Queue`	Yes	No	No
`ConcurrentSkipListMap`	`ConcurrentMap` `SortedMap` `NavigableMap`	Yes	Yes	No
`ConcurrentSkipListSet`	`SortedSet` `NavigableSet`	Yes	Yes	No
`CopyOnWriteArrayList`	`List`	Yes	No	No
`CopyOnWriteArraySet`	`Set`	No	No	No
`LinkedBlockingQueue`	`BlockingQueue`	Yes	No	Yes

Based on your knowledge of collections from Chapter 14, classes like `ConcurrentHashMap` and `ConcurrentLinkedQueue` should be quite easy for you to learn. Take a look at the following code samples:

```
Map<String,Integer> map = new ConcurrentHashMap<>();
map.put("zebra", 52);
```

```
map.put("elephant", 10);
System.out.println(map.get("elephant"));  // 10

Queue<Integer> queue = new ConcurrentLinkedQueue<>();
queue.offer(31);
System.out.println(queue.peek());  // 31
System.out.println(queue.poll());  // 31
```

Like we often did in Chapter 14, we use an interface reference for the variable type of the newly created object and use it the same way as we would a nonconcurrent object. The difference is that these objects are safe to pass to multiple threads.

All of these classes implement multiple interfaces. For example, `ConcurrentHashMap` implements `Map` and `ConcurrentMap`. When declaring methods that take a concurrent collection, it is up to you to determine the appropriate method parameter type. For example, a method signature may require a `ConcurrentMap` reference to ensure that an object passed to it is properly supported in a multithreaded environment.

Understanding *SkipList* Collections

The `SkipList` classes, `ConcurrentSkipListSet` and `ConcurrentSkipListMap`, are concurrent versions of their sorted counterparts, `TreeSet` and `TreeMap`, respectively. They maintain their elements or keys in the natural ordering of their elements. In this manner, using them is the same as the code that you worked with in Chapter 14.

```
Set<String> gardenAnimals = new ConcurrentSkipListSet<>();
gardenAnimals.add("rabbit");
gardenAnimals.add("gopher");
System.out.println(gardenAnimals.stream()
   .collect(Collectors.joining(",")));  // gopher,rabbit

Map<String, String> rainForestAnimalDiet
   = new ConcurrentSkipListMap<>();
rainForestAnimalDiet.put("koala", "bamboo");
rainForestAnimalDiet.entrySet()
   .stream()
```

```
      .forEach((e) -> System.out.println(
         e.getKey() + "-" + e.getValue())); // koala-bamboo
```

When you see `SkipList` or `SkipSet` on the exam, just think "sorted" concurrent collections, and the rest should follow naturally.

Understanding *CopyOnWrite* Collections

Table 18.9 included two classes, `CopyOnWriteArrayList` and `CopyOnWriteArraySet`, that behave a little differently than the other concurrent examples that you have seen. These classes copy all of their elements to a new underlying structure anytime an element is added, modified, or removed from the collection. By a *modified* element, we mean that the reference in the collection is changed. Modifying the actual contents of objects within the collection will not cause a new structure to be allocated.

Although the data is copied to a new underlying structure, our reference to the `Collection` object does not change. This is particularly useful in multithreaded environments that need to iterate the collection. Any iterator established prior to a modification will not see the changes, but instead it will iterate over the original elements prior to the modification.

Let's take a look at how this works with an example. Does the following program terminate? If so, how many times does the loop execute?

```
List<Integer> favNumbers =
   new CopyOnWriteArrayList<>(List.of(4,3,42));
for(var n: favNumbers) {
   System.out.print(n + " ");
   favNumbers.add(9);
}
System.out.println();
System.out.println("Size: " + favNumbers.size());
```

When executed as part of a program, this code snippet outputs the following:

```
4 3 42
Size: 6
```

Despite adding elements to the array while iterating over it, the `for` loop only iterated on the ones created when the loop started. Alternatively, if we had used a regular `ArrayList` object, a `ConcurrentModificationException` would have been thrown at run-time. With either class, though, we avoid entering an infinite loop in which elements are constantly added to the array as we iterate over them.

The `CopyOnWrite` classes are similar to the immutable object pattern that you saw in Chapter 12, "Java Fundamentals," as a new underlying structure is created every time the collection is modified. Unlike a true immutable object, though, the reference to the object stays the same even while the underlying data is changed.

The `CopyOnWriteArraySet` is used just like a `HashSet` and has similar properties as the `CopyOnWriteArrayList` class.

```
Set<Character> favLetters =
   new CopyOnWriteArraySet<>(List.of('a','t'));
for(char c: favLetters) {
   System.out.print(c+" ");
   favLetters.add('s');
}
System.out.println();
System.out.println("Size: "+ favLetters.size());
```

This code snippet prints:

```
a t
Size: 3
```

The `CopyOnWrite` classes can use a lot of memory, since a new collection structure needs be allocated anytime the collection is modified. They are commonly used in multithreaded environment situations where reads are far more common than writes.

REVISITING DELETING WHILE LOOPING

In Chapter 14, we showed an example where deleting from an `ArrayList` while iterating over it triggered a `ConcurrentModificationException`. Here we present a version that does work using `CopyOnWriteArrayList`:

```
List<String> birds = new CopyOnWriteArrayList<>();
birds.add("hawk");
birds.add("hawk");
birds.add("hawk");

for (String bird : birds)
    birds.remove(bird);
System.out.print(birds.size()); // 0
```

As mentioned, though, `CopyOnWrite` classes can use a lot of memory. Another approach is to use the `ArrayList` class with an iterator, as shown here:

```
    var iterator = birds.iterator();
    while(iterator.hasNext()) {
       iterator.next();
       <b>iterator.remove()</b>;
    }
    System.out.print(birds.size());  // 0
```

Understanding Blocking Queues

The final collection class in Table 18.9 that you should know for the exam is the `LinkedBlockingQueue`, which implements the `BlockingQueue` interface. The `BlockingQueue` is just like a regular `Queue`, except that it includes methods that will wait a specific amount of time to complete an operation.

Since `BlockingQueue` inherits all of the methods from `Queue`, we skip the inherited methods you learned in Chapter 14 and present the new methods in Table 18.10.

TABLE 18.10 `BlockingQueue` waiting methods

Method name	Description
`offer(E e, long timeout, TimeUnit unit)`	Adds an item to the queue, waiting the specified time and returning `false` if the time elapses before space is available
`poll(long timeout, TimeUnit unit)`	Retrieves and removes an item from the queue, waiting the specified time and returning `null` if the time elapses before the item is available

The implementation class `LinkedBlockingQueue`, as the name implies, maintains a linked list between elements. The following sample is using a `LinkedBlockingQueue` to wait for the results of some of the operations. The methods in Table 18.10 can each throw a checked `InterruptedException`, as they can be interrupted before they finish waiting for a result; therefore, they must be properly caught.

```
try {
   var blockingQueue = new LinkedBlockingQueue<Integer>();
   blockingQueue.offer(39);
   blockingQueue.offer(3, 4, TimeUnit.SECONDS);
   System.out.println(blockingQueue.poll());
```

```
      System.out.println(blockingQueue.poll(10, TimeUnit.MILLISECONDS));
   } catch (InterruptedException e) {
      // Handle interruption
   }
```

This code snippet prints the following:

```
39
3
```

As shown in this example, since `LinkedBlockingQueue` implements both `Queue` and `BlockingQueue`, we can use methods available to both, such as those that don't take any wait arguments.

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 methods in Table 18.11.

TABLE 18.11 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)`

When should you use these methods? If you know at the time of creation that your object requires synchronization, then you should use one of the concurrent collection classes listed in Table 18.9. On the other hand, if you are given an existing collection that is not a concurrent class and need to access it among multiple threads, you can wrap it using the methods in Table 18.11.

Unlike the concurrent collections, the synchronized collections also throw an exception if they are modified within an iterator by a single thread. For example, take a look at the following modification of our earlier example:

```
var foodData = new HashMap<String, Object>();
foodData.put("penguin", 1);
foodData.put("flamingo", 2);
var synFoodData = Collections.synchronizedMap(foodData);
for(String key: synFoodData.keySet())
   synFoodData.remove(key);
```

This loop throws a `ConcurrentModificationException`, whereas our example that used `ConcurrentHashMap` did not. Other than iterating over the collection, the objects returned by the methods in Table 18.11 are safe from memory consistency errors and can be used among multiple threads.

Identifying Threading Problems

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 it. 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 in some kind of "stuck" state. 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:

```
import java.util.concurrent.*;
class Food {}
class Water {}
public class Fox {
   public void eatAndDrink(Food food, Water water) {
      synchronized(food) {
         System.out.println("Got Food!");
         move();
         synchronized(water) {
            System.out.println("Got Water!");
         }
      }
   }
   public void drinkAndEat(Food food, Water water) {
      synchronized(water) {
         System.out.println("Got Water!");
         move();
```

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

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:

```
Got Food!
Got Water!
```

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

PREVENTING DEADLOCKS

How do you fix a deadlock once it has occurred? The answer is that you can't in most situations. On the other hand, there are numerous strategies to help prevent deadlocks from ever happening in the first place. One common strategy to avoid deadlocks is for all threads to order their resource requests. For example, if both foxes have a rule that they need to obtain food before water, then the previous deadlock scenario will not happen again. Once one of the foxes obtained food, the second fox would wait, leaving the water resource available.

There are some advanced techniques that try to detect and resolve a deadlock in real time, but they are often quite difficult to implement and have limited success in practice. In fact, many operating systems ignore the problem altogether and pretend that deadlocks never happen.

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 they are 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 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 is able to 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 in fact stuck in an endless cycle.

Managing Race Conditions

A *race condition* is an undesirable result that occurs when two tasks, which 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 18.3 shows a classical thread-based example of a race condition, we now provide a more illustrative example. Imagine 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 they each send requests to create the account at the same time, as shown in Figure 18.5.

What result does the web server return when both users attempt to create an account with the same username in Figure 18.5?

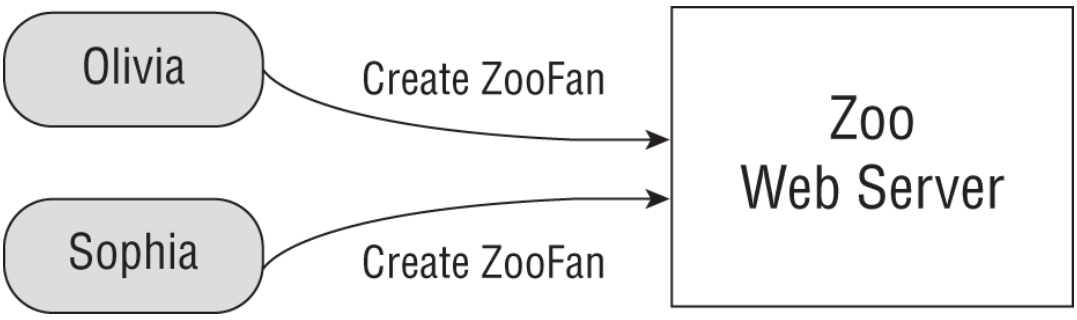

FIGURE 18.5 Race condition on user creation

Possible Outcomes for This Race Condition

- Both users are able to create accounts with username ZooFan.
- Both users are unable to create an account with username ZooFan, returning an error message to both users.
- One user is able to create the account with the username ZooFan, while the other user receives an error message.

Which of these results is most desirable when designing our web server? The first possibility, in which both users are able to create an account with the same username, could cause serious problems and break numerous invariants in the system. Assuming that the username is required to log into the website, how do they both log in with the same username and different passwords? In this case, the website cannot tell them apart. This is the worst possible outcome to this race condition, as it causes significant and potentially unrecoverable data problems.

What about the second scenario? If both users are unable to create the account, both will receive error messages and be told to try again. In this scenario, the data is protected since no two accounts with the same username exist in the system. The users are free to try again with the same username, ZooFan, since no one has been granted access to it. Although this might seem like a form of livelock, there is a subtle difference. When the users try to create their account again, the chances of them hitting a race condition tend to diminish. For example, if one user submits their request a few seconds before the other, they might avoid another race con-

dition entirely by the system informing the second user that the account name is already in use.

The third scenario, in which one user obtains the account while the other does not, is often considered the best solution to this type of race condition. 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. Also unlike the previous scenario, we can provide the user who didn't win the race with a clearer error message because we are now sure that the account username is no longer available in the system.

For the third scenario, which of the two users should gain access to the account? For race conditions, it often doesn't matter as long as only one player “wins” the race. A common practice is to choose whichever thread made the request first, whenever possible.

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.

Race conditions tend to appear in highly concurrent applications. As a software system grows and more users are added, they tend to appear more frequently. One solution is to use a monitor to synchronize on the relevant overlapping task. In the previous example, the relevant task is the method that determines whether an account username is in use and reserves it in the system if it is available.

Working with Parallel Streams

One of the most powerful features of the Stream API is built-in concurrency support. Up until now, all of the streams with which you have worked 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 a stream that 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.

The number of threads available in a parallel stream is proportional to the number of available CPUs in your environment.

Creating Parallel Streams

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

Calling *parallel()* on an Existing Stream

The first way to create a parallel stream is from an existing stream. You just call `parallel()` on an existing stream to convert it to one that supports multithreaded processing, as shown in the following code:

```
Stream<Integer> s1 = List.of(1,2).stream();
Stream<Integer> s2 = s1.parallel();
```

Be aware that `parallel()` is an intermediate operation that operates on the original stream. For example, applying a terminal operation to `s2` also makes `s1` unavailable for further use.

Calling *parallelStream()* on a *Collection* Object

The second way to create a parallel stream is from a Java `Collection` class. The `Collection` interface includes a method `parallelStream()` that can be called on any collection and returns a parallel stream. The following creates the parallel stream directly from the `List` object:

```
Stream<Integer> s3 = List.of(1,2).parallelStream();
```

We will use both `parallel()` and `parallelStream()` throughout this section.

The `Stream` interface includes a method `isParallel()` that can be used to test if the instance of a stream supports parallel processing. Some operations on streams preserve the parallel attribute, while others do not. For example, the `Stream.concat(Stream s1, Stream s2)` is parallel if either `s1` or `s2` is parallel. On the other hand, `flatMap()` creates a new stream that is not parallel by default, regardless of whether the underlying elements were parallel.

Performing a Parallel Decomposition

As you may have noticed, creating the parallel stream is the easy part. The interesting part comes in performing a parallel decomposition. A *parallel decomposition* is the process of taking a task, breaking it up 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. For starters, 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 might be calling a database or reading a file. Now let's use this method with a serial stream.

```
long start = System.currentTimeMillis();
List.of(1,2,3,4,5)
   .stream()
   .map(w -> doWork(w))
   .forEach(s -> System.out.print(s + " "));

System.out.println();
var timeTaken = (System.currentTimeMillis()-start)/1000;
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 use a parallel stream, though?

```
long start = System.currentTimeMillis();
List.of(1,2,3,4,5)
   .parallelStream()
   .map(w -> doWork(w))
   .forEach(s -> System.out.print(s + " "));
```

```
System.out.println();
var timeTaken = (System.currentTimeMillis()-start)/1000;
System.out.println("Time: "+timeTaken+" seconds");
```

With a parallel stream, the `map()` and `forEach()` operations are applied concurrently. The following is 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 *FOREACH* RESULTS

The Stream API includes an alternate version of the `forEach()` operation called `forEachOrdered()`, which forces a parallel stream to process the results in order at the cost of performance. For example, take a look at the following code snippet:

```
List.of(5,2,1,4,3)
   .parallelStream()
   .map(w -> doWork(w))
   .forEachOrdered(s -> System.out.print(s + " "));
```

Like our starting example, this outputs the results in the order that they are defined in the stream:

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

With this change, the `forEachOrdered()` operation forces our stream into a single-threaded process. While we've lost some of the performance gains of using a parallel stream, our `map()` operation is still able to take advantage of the parallel stream and perform a parallel decomposition in 5 seconds instead of 25 seconds.

Processing Parallel Reductions

Besides possibly improving performance and modifying the order of operations, using parallel streams can impact how you write your application. Reduction operations on parallel streams are referred to as *parallel reductions*. The results for parallel reductions can be different 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. Let's

take a look at the results of `findAny()` applied to a serial stream.

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

This code frequently outputs the first value in the serial stream, `1`, although this is not guaranteed. The `findAny()` method is free to select any element on either serial or parallel streams.

With a parallel stream, the JVM can create any number of threads to process the stream. When you call `findAny()` on a parallel stream, the JVM selects the first thread to finish the task and retrieves its data.

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

The result is that the output could be `4`, `1`, or really any value in the stream. You can see that with parallel streams, the results of `findAny()` are not as predictable.

Any stream operation that is based on order, including `findFirst()`, `limit()`, or `skip()`, may actually perform more slowly in a parallel environment. This is a result of a parallel processing task being forced to coordinate all of its threads in a synchronized-like fashion.

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 with which you have been working 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 actually 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 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 15, 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 work regardless of the order they are called in. For example, if we add numbers, we can do so in any order.

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()` do 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`, then 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 are unable to be completed in parallel.

Performing a Parallel Reduction on a *Collector*

While we covered the `Collector` interface in Chapter 15, 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:

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

The `Collectors` class includes two sets of `static` methods for retrieving collectors, `toConcurrentMap()` and `groupingByConcurrent()`, that are both `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 15 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.ConcurrentHashM
```

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 15 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 a `ConcurrentMap` reference.

ENCOURAGING PARALLEL PROCESSING

Guaranteeing that a particular stream will perform reductions in parallel, as opposed to single-threaded, is often difficult in practice. For example, the one-argument `reduce()` operation on a parallel stream may perform concurrently even when there is no explicit combiner argument. Alternatively, you may expect some collectors to perform well on a parallel stream, resorting to single-threaded processing at runtime.

The key to applying parallel reductions is to encourage the JVM to take advantage of the parallel structures, such as using a `groupingByConcurrent()` collector on a parallel stream rather than a `groupingBy()` collector. By encouraging the JVM to take advantage of the parallel processing, we get the best possible performance at runtime.

Avoiding Stateful Operations

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. On the other hand, a *stateless lambda expression* is one whose result does not depend on any state that might change during the execution of a pipeline.

Let's try an example. Imagine we require a method that keeps only even numbers in a stream and adds them to a list. Also, we want ordering of the numbers in the stream and list to be consistent. The following `addValues()` method accomplishes this:

```
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 the following stream:

```
var list = addValues(IntStream.range(1, 11));
System.out.println(list);
```

Then, the output would be as follows:

```
[2, 4, 6, 8, 10]
```

But what if someone else wrote an implementation that passed our method a parallel stream?

```
var list = addValues(IntStream.range(1, 11).parallel());
System.out.println(list);
```

With a parallel stream, the order of the output becomes random.

```
[6, 8, 10, 2, 4]
```

The problem is that our lambda expression is stateful and modifies a list that is outside our stream. We could use `forEachOrdered()` to add elements to the list, but that forces the parallel stream to be serial, potentially losing concurrency enhancements. While these stream operations in our example are quite simple, imagine using them alongside numerous intermediate operations.

We can fix this solution by rewriting our stream operation to no longer have a stateful lambda expression.

```
public static List<Integer> addValues(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 result on both serial and parallel streams.

```
[2, 4, 6, 8, 10]
```

This implementation removes the stateful operation and relies on the collector to assemble the elements. We could also use a concurrent collector to parallelize the building of the list. The goal is to write our code to allow for parallel processing and let the JVM handle the rest.

It is strongly recommended that you avoid stateful operations when using parallel streams, so as 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 threads and showed you how to process tasks in parallel using the Concurrency API. The work that a thread performs can be expressed as lambda expressions or instances of `Runnable` or `Callable`.

For the exam, you should know how to concurrently execute tasks using `ExecutorService`. 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 collections classes that handle multithreaded access for you. For the exam, you should also be familiar with the concurrent collections including the `CopyOnWriteArrayList` class, which creates a new underlying structure anytime the list 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 often quite challenging.

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 results are no longer ordered. Remember to avoid stateful lambda expressions, especially when working with parallel streams.

Exam Essentials

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 a fixed interval between executions.

Be able to apply the atomic classes. An atomic operation is one that occurs without interference by 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.

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 on it, as well as supporting fair acquisi-

tion of locks. To achieve synchronization, two threads must synchronize on the same shared object.

Manage a process with a *CyclicBarrier*. The `CyclicBarrier` class can be used to force a set of threads to wait until they are at a certain stage of execution before continuing.

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 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 an instance of a `Stream s` and a `Collection c`, which are valid ways of creating a parallel stream? (Choose all that apply.)

A. `new ParallelStream(s)`

B. `c.parallel()`

C. `s.parallelStream()`

D. `c.parallelStream()`

E. `new ParallelStream(c)`

F. `s.parallel()`

2. Given that the sum of the numbers from 1 (inclusive) to 10 (exclusive) is 45, what are the possible results of executing the following program? (Choose all that apply.)

```
    import java.util.concurrent.locks.*;
import java.util.stream.*;
public class Bank {
   private Lock vault = new ReentrantLock();
   private int total = 0;
   public void deposit(int value) {
      try {
         vault.tryLock();
         total += value;
      } finally {
         vault.unlock();
      }
   }
   public static void main(String[] unused) {
      var bank = new Bank();
      IntStream.range(1, 10).parallel()
         .forEach(s -> bank.deposit(s));
      System.out.println(bank.total);
   } }
```

A. `45` is printed.

B. A number less than `45` is printed.

C. A number greater than `45` is printed.

D. An exception is thrown.

E. None of the above, as the code does not compile

3. Which of the following statements about the `Callable call()` and `Runnable run()` methods are correct? (Choose all that apply.)

A. Both can throw unchecked exceptions.

B. `Callable` takes a generic method argument.

C. `Callable` can throw a checked exception.

D. Both can be implemented with lambda expressions.

E. `Runnable` returns a generic type.

F. `Callable` returns a generic type.

G. Both methods return `void`.

4. Which lines need to be changed to make the code compile? (Choose all that apply.)

```
ExecutorService service =   // w1
   Executors.newSingleThreadScheduledExecutor();
service.scheduleWithFixedDelay(() -> {
   System.out.println("Open Zoo");
   return null;   // w2
}, 0, 1, TimeUnit.MINUTES);
var result = service.submit(() ->   // w3
   System.out.println("Wake Staff"));
System.out.println(result.get());   // w4
```

A. It compiles and runs without issue.

B. Line `w1`

C. Line `w2`

D. Line `w3`

E. Line `w4`

F. It compiles but throws an exception at runtime.

G. None of the above

5. What statement about the following code is true?

```
var value1 = new AtomicLong(0);
final long[] value2 = {0};
IntStream.iterate(1, i -> 1).limit(100).parallel()
   .forEach(i -> value1.incrementAndGet());
IntStream.iterate(1, i -> 1).limit(100).parallel()
   .forEach(i -> ++value2[0]);
System.out.println(value1+" "+value2[0]);
```

A. It outputs `100 100`.

B. It outputs `100 99`.

C. The output cannot be determined ahead of time.

D. The code does not compile.

E. It compiles but throws an exception at runtime.

F. It compiles but enters an infinite loop at runtime.

G. None of the above

6. Which statements about the following code are correct? (Choose all that apply.)

```
public static void main(String[] args) throws Exception {
   var data = List.of(2,5,1,9,8);
   data.stream().parallel()
      .mapToInt(s -> s)
      .peek(System.out::println)
      .forEachOrdered(System.out::println);
}
```

A. The `peek()` method will print the entries in the order: `1 2 5 8 9`.

B. The `peek()` method will print the entries in the order: `2 5 1 9 8`.

C. The `peek()` method will print the entries in an order that cannot be determined ahead of time.

D. The `forEachOrdered()` method will print the entries in the order: `1 2 5 8 9`.

E. The `forEachOrdered()` method will print the entries in the order: `2 5 1 9 8`.

F. The `forEachOrdered()` method will print the entries in an order that cannot be determined ahead of time.

G. The code does not compile.

7. Fill in the blanks: _____ occur(s) when two or more threads are blocked forever but both appear active. _____ occur(s) when two or more threads try to complete a related task at the same time, resulting in invalid or unexpected data.

A. Livelock, Deadlock

B. Deadlock, Starvation

C. Race conditions, Deadlock

D. Livelock, Race conditions

E. Starvation, Race conditions

F. Deadlock, Livelock

8. Assuming this class is accessed by only a single thread at a time, what is the result of calling the `countIceCreamFlavors()` method?

```
import java.util.stream.LongStream;
public class Flavors {
   private static int counter;
   public static void countIceCreamFlavors()  {
      counter = 0;
      Runnable task = () -> counter++;
      LongStream.range(1, 500)
         .forEach(m -> new Thread(task).run());
      System.out.println(counter);
   }
}
```

   A. The method consistently prints `499`.
   B. The method consistently prints `500`.
   C. The method compiles and prints a value, but that value cannot be determined ahead of time.
   D. The method does not compile.
   E. The method compiles but throws an exception at runtime.
   F. None of the above

9. Which happens when a new task is submitted to an `ExecutorService`, in which there are no threads available?
   A. The executor throws an exception when the task is submitted.
   B. The executor discards the task without completing it.
   C. The executor adds the task to an internal queue and completes when there is an available thread.
   D. The thread submitting the task waits on the submit call until a thread is available before continuing.
   E. The executor creates a new temporary thread to complete the task.

10. What is the result of executing the following code snippet?

```
List<Integer> lions = new ArrayList<>(List.of(1,2,3));
List<Integer> tigers = new CopyOnWriteArrayList<>(lions);
Set<Integer> bears = new ConcurrentSkipListSet<>();
bears.addAll(lions);
for(Integer item: tigers) tigers.add(4); // x1
```

```
for(Integer item: bears) bears.add(5); // x2
System.out.println(lions.size() + " " + tigers.size()
   + " " + bears.size());
```

A. It outputs `3 6 4`.
B. It outputs `6 6 6`.
C. It outputs `6 3 4`.
D. The code does not compile.
E. It compiles but throws an exception at runtime on line `x1`.
F. It compiles but throws an exception at runtime on line `x2`.
G. It compiles but enters an infinite loop at runtime.

11. What statements about the following code are true? (Choose all that apply.)

```
Integer i1 = List.of(1, 2, 3, 4, 5).stream().findAny().get();
synchronized(i1) { // y1
   Integer i2 = List.of(6, 7, 8, 9, 10)
      .parallelStream()
      .sorted()
      .findAny().get(); // y2
   System.out.println(i1 + " " + i2);
}
```

A. The first value printed is always `1`.
B. The second value printed is always `6`.
C. The code will not compile because of line `y1`.
D. The code will not compile because of line `y2`.
E. The code compiles but throws an exception at runtime.
F. The output cannot be determined ahead of time.
G. It compiles but waits forever at runtime.

12. Assuming `takeNap()` is a method that takes five seconds to execute without throwing an exception, what is the expected result of executing the following code snippet?

```
ExecutorService service = null;
try {
   service = Executors.newFixedThreadPool(4);
```

```
      service.execute(() -> takeNap());
      service.execute(() -> takeNap());
      service.execute(() -> takeNap());
   } finally {
      if (service != null) service.shutdown();
   }
   service.awaitTermination(2, TimeUnit.SECONDS);
   System.out.println("DONE!");
```

A. It will immediately print `DONE!`.
B. It will pause for 2 seconds and then print `DONE!`.
C. It will pause for 5 seconds and then print `DONE!`.
D. It will pause for 15 seconds and then print `DONE!`.
E. It will throw an exception at runtime.
F. None of the above, as the code does not compile

13. What statements about the following code are true? (Choose all that apply.)

```
System.out.print(List.of("duck","flamingo","pelican")
   .parallelStream().parallel()   // q1
   .reduce(0,
      (c1, c2) -> c1.length() + c2.length(),   // q2
      (s1, s2) -> s1 + s2));   // q3
```

A. It compiles and runs without issue, outputting the total length of all strings in the stream.
B. The code will not compile because of line `q1`.
C. The code will not compile because of line `q2`.
D. The code will not compile because of line `q3`.
E. It compiles but throws an exception at runtime.
F. None of the above

14. What statements about the following code snippet are true? (Choose all that apply.)

```
Object o1 = new Object();
Object o2 = new Object();
var service = Executors.newFixedThreadPool(2);
var f1 = service.submit(() -> {
```

```
      synchronized (o1) {
         synchronized (o2) { System.out.print("Tortoise"); }
      }
   });
   var f2 = service.submit(() -> {
      synchronized (o2) {
         synchronized (o1) { System.out.print("Hare"); }
      }
   });
   f1.get();
   f2.get();
```

A. The code will always output `Tortoise` followed by `Hare`.
B. The code will always output `Hare` followed by `Tortoise`.
C. If the code does output anything, the order cannot be determined.
D. The code does not compile.
E. The code compiles but may produce a deadlock at runtime.
F. The code compiles but may produce a livelock at runtime.
G. It compiles but throws an exception at runtime.

15. Which statement about the following code snippet is correct?

```
2: var cats = Stream.of("leopard", "lynx", "ocelot", "puma")
3:    .parallel();
4: var bears = Stream.of("panda","grizzly","polar").parallel();
5: var data = Stream.of(cats,bears).flatMap(s -> s)
6:    .collect(Collectors.groupingByConcurrent(
7:       s -> !s.startsWith("p")));
8: System.out.println(data.get(false).size()
9:    + " " + data.get(true).size());
```

A. It outputs `3 4`.
B. It outputs `4 3`.
C. The code will not compile because of line 6.
D. The code will not compile because of line 7.
E. The code will not compile because of line 8.
F. It compiles but throws an exception at runtime.

16. Which statements about methods in `ReentrantLock` are correct? (Choose all that apply.)

A. The `lock()` method will attempt to acquire a lock without waiting indefinitely for it.
B. The `testLock()` method will attempt to acquire a lock without waiting indefinitely for it.
C. The `attemptLock()` method will attempt to acquire a lock without waiting indefinitely for it.
D. By default, a `ReentrantLock` fairly releases to each thread, in the order that it was requested.
E. Calling the `unlock()` method once will release a resource so that other threads can obtain the lock.
F. None of the above

17. What is the result of calling the following method?

```
3: public void addAndPrintItems(BlockingQueue<Integer> queue) {
4:    queue.offer(103);
5:    queue.offer(20, 1, TimeUnit.SECONDS);
6:    queue.offer(85, 7, TimeUnit.HOURS);
7:    System.out.print(queue.poll(200, TimeUnit.NANOSECONDS));
8:    System.out.print(" " + queue.poll(1, TimeUnit.MINUTES));
9: }
```

A. It outputs `20 85`.
B. It outputs `103 20`.
C. It outputs `20 103`.
D. The code will not compile.
E. It compiles but throws an exception at runtime.
F. The output cannot be determined ahead of time.
G. None of the above

18. Which of the following are valid `Callable` expressions? (Choose all that apply.)

A. `a -> {return 10;}`
B. `() -> {String s = "";}`
C. `() -> 5`
D. `() -> {return null}`
E. `() -> "The" + "Zoo"`
F. `(int count) -> count+1`
G. `() -> {System.out.println("Giraffe"); return 10;}`

19. What is the result of executing the following application? (Choose all that apply.)

```
import java.util.concurrent.*;
import java.util.stream.*;
public class PrintConstants {
   public static void main(String[] args) {
      var s = Executors.newScheduledThreadPool(10);
      DoubleStream.of(3.14159,2.71828)   // b1
         .forEach(c -> s.submit(  // b2
            () -> System.out.println(10*c)));   // b3
      s.execute(() -> System.out.println("Printed"));   // b4
   }
}
```

A. It compiles and outputs the two numbers, followed by `Printed`.
B. The code will not compile because of line `b1`.
C. The code will not compile because of line `b2`.
D. The code will not compile because of line `b3`.
E. The code will not compile because of line `b4`.
F. It compiles, but the output cannot be determined ahead of time.
G. It compiles but throws an exception at runtime.
H. It compiles but waits forever at runtime.

20. What is the result of executing the following program? (Choose all that apply.)

```
import java.util.*;
import java.util.concurrent.*;
import java.util.stream.*;
public class PrintCounter {
   static int count = 0;
   public static void main(String[] args) throws
                     InterruptedException, ExecutionException {
      ExecutorService service = null;
      try {
         service = Executors.newSingleThreadExecutor();
         var r = new ArrayList<Future<?>>();
         IntStream.iterate(0,i -> i+1).limit(5).forEach(
            i -> r.add(service.execute(() -> {count++;})) // n1
```

```
            );
            for(Future<?> result : r) {
               System.out.print(result.get()+" "); // n2
            }
         } finally {
            if(service != null) service.shutdown();
         } } }
```

A. It prints `0 1 2 3 4`.
B. It prints `1 2 3 4 5`.
C. It prints `null null null null null`.
D. It hangs indefinitely at runtime.
E. The output cannot be determined.
F. The code will not compile because of line `n1`.
G. The code will not compile because of line `n2`.

21. Given the following code snippet and blank lines on `p1` and `p2`, which values guarantee `1` is printed at runtime? (Choose all that apply.)

```
var data = List.of(List.of(1,2),
   List.of(3,4),
   List.of(5,6));
data. _____________ // p1
   .flatMap(s -> s.stream())
   . ____________ // p2
   .ifPresent(System.out::print);
```

A. `stream()` on line `p1`, `findFirst()` on line `p2`.
B. `stream()` on line `p1`, `findAny()` on line `p2`.
C. `parallelStream()` in line `p1`, `findAny()` on line `p2`.
D. `parallelStream()` in line `p1`, `findFirst()` on line `p2`.
E. The code does not compile regardless of what is inserted into the blank.
F. None of the above

22. Assuming 100 milliseconds is enough time for the tasks submitted to the service executor to complete, what is the result of executing the following method? (Choose all that apply.)

```
private AtomicInteger s1 = new AtomicInteger(0); // w1
private int s2 = 0;

private void countSheep() throws InterruptedException {
   ExecutorService service = null;
   try {
      service = Executors.newSingleThreadExecutor(); // w2
      for (int i = 0; i < 100; i++)
         service.execute(() -> {
            s1.getAndIncrement(); s2++; }); // w3
      Thread.sleep(100);
      System.out.println(s1 + " " + s2);
   } finally {
      if(service != null) service.shutdown();
   }
}
```

A. The method consistently prints `100 99`.
B. The method consistently prints `100 100`.
C. The output cannot be determined ahead of time.
D. The code will not compile because of line `w1`.
E. The code will not compile because of line `w2`.
F. The code will not compile because of line `w3`.
G. It compiles but throws an exception at runtime.

23. What is the result of executing the following application? (Choose all that apply.)

```
import java.util.concurrent.*;
import java.util.stream.*;
public class StockRoomTracker {
   public static void await(CyclicBarrier cb) { // j1
      try { cb.await(); } catch (Exception e) {}
   }
   public static void main(String[] args) {
      var cb = new CyclicBarrier(10,
         () -> System.out.println("Stock Room Full!")); // j2
      IntStream.iterate(1, i -> 1).limit(9).parallel()
         .forEach(i -> await(cb)); // j3
   }
}
```

A. It outputs `Stock Room Full!`

B. The code will not compile because of line `j1`.

C. The code will not compile because of line `j2`.

D. The code will not compile because of line `j3`.

E. It compiles but throws an exception at runtime.

F. It compiles but waits forever at runtime.

24. What statements about the following class definition are true? (Choose all that apply.)

```
public class TicketManager {
   private int tickets;
   private static TicketManager instance;
   private TicketManager() {}
   static synchronized TicketManager getInstance() {      // k1
      if (instance==null) instance = new TicketManager(); // k2
      return instance;
   }

   public int getTicketCount() { return tickets; }
   public void addTickets(int value) {tickets += value;}  // k3
   public void sellTickets(int value) {
      synchronized (this) { // k4
         tickets -= value;
      }
   }
}
```

A. It compiles without issue.

B. The code will not compile because of line `k2`.

C. The code will not compile because of line `k3`.

D. The locks acquired on `k1` and `k4` are on the same object.

E. The class correctly protects the `tickets` data from race conditions.

F. At most one instance of `TicketManager` will be created in an application that uses this class.

25. Which of the following properties of concurrency are true? (Choose all that apply.)

A. By itself, concurrency does not guarantee which task will be completed first.
B. Concurrency always improves the performance of an application.
C. A computer with a single-processor CPU does not benefit from concurrency.
D. Applications with many resource-heavy tasks tend to benefit more from concurrency than ones with CPU-intensive tasks.
E. Concurrent tasks do not share the same memory.

26. Assuming an implementation of the `performCount()` method is provided prior to runtime, which of the following are possible results of executing the following application? (Choose all that apply.)

```
import java.util.*;
import java.util.concurrent.*;
public class CountZooAnimals {
   public static void performCount(int animal) {
      // IMPLEMENTATION OMITTED
   }
   public static void printResults(Future<?> f) {
      try {
         System.out.println(f.get(1, TimeUnit.DAYS)); // o1
      } catch (Exception e) {
         System.out.println("Exception!");
      }
   }
   public static void main(String[] args) throws Exception {
      ExecutorService s = null;
      final var r = new ArrayList<Future<?>>();
      try {
         s = Executors.newSingleThreadExecutor();
         for(int i = 0; i < 10; i++) {
            final int animal = i;
            r.add(s.submit(() -> performCount(animal))); // o2
         }
         r.forEach(f -> printResults(f));
      } finally {
         if(s != null) s.shutdown();
      } } }
```

A. It outputs a number 10 times.

B. It outputs a `Boolean` value 10 times.

C. It outputs a `null` value 10 times.

D. It outputs `Exception!` 10 times.

E. It hangs indefinitely at runtime.

F. The code will not compile because of line `o1`.

G. The code will not compile because of line `o2`.