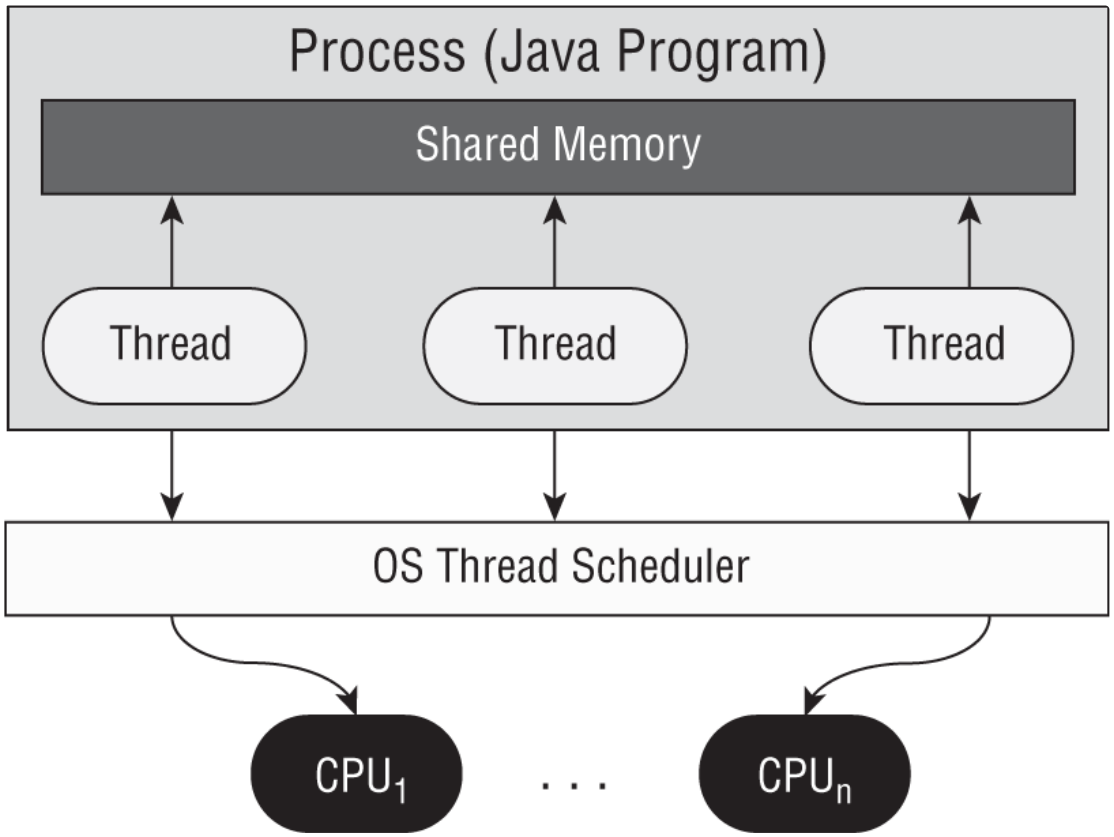
Chapter 18 - Concurrency

The idea behind multithreaded processing is to allow an application or group of applications to execute multiple tasks at the same time.

Since its early days, Java has supported multithreaded programming using the `Thread` class.

Introducing Threads

- A *thread* is the smallest unit of execution that can be scheduled by the operating system.
- A *process* is a group of associated threads that execute in the same, shared environment. It follows, then, that a ***single-threaded process*** is one that contains exactly one thread, whereas a ***multithreaded*** *process* 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.
- A *task* is a single unit of work performed by a thread. A thread can complete multiple independent tasks but only one task at a time.



Distinguishing Thread Types

- 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`.
- 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.

Understanding Thread Concurrency

Operating systems use a *thread scheduler* to determine which threads should be currently executing. For example, a thread scheduler may employ a *round-robin schedule* in which each available thread receives an equal number of CPU cycles with which to execute, with threads visited in a circular order. 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.

A *context switch* is the process of storing a thread's current state and later restoring the state of the thread to continue execution.

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.

Defining a Task with *Runnable*

`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 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 = () -> {};
```

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
       }
    }
```

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.

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();
   }
}
```

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.

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.

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.

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.

Introducing the Single-Thread Executor

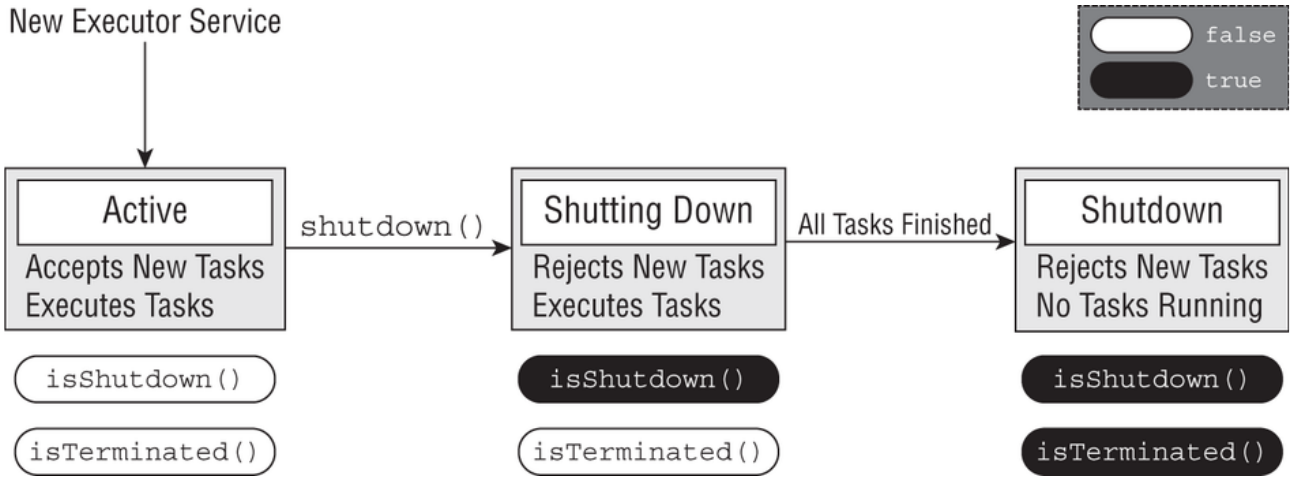
Since `ExecutorService` is an interface, how do you obtain an instance of it? The Concurrency API includes the `Executors` factory class that can be used to create instances of the `ExecutorService` object.

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

```
1  import java.util.concurrent.*;
2  public class ZooInfo {
3     public static void main(String[] args) {
4        ExecutorService service = null;
5        Runnable task1 = () ->
6           System.out.println("Printing zoo inventory");
7        Runnable task2 = () -> {for(int i = 0; i < 3; i++)
8              System.out.println("Printing record: "+i);};
9        try {
10          service = Executors.newSingleThreadExecutor();
11          System.out.println("begin");
12          service.execute(task1);
13          service.execute(task2);
14          service.execute(task1);
15          System.out.println("end");
16       } finally {
17          if(service != null) service.shutdown();
18       }
19    }
20 }
```

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.



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.

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 `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

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.

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.

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:

```
1  @FunctionalInterface public interface Callable<V> {
2     V call() throws Exception;
3  }
```

Waiting for All Tasks to Finish

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

```
1  ExecutorService service = null;
2  try {
3     service = Executors.newSingleThreadExecutor();
4     // Add tasks to the thread executor
5     …
6  } finally {
7     if(service != null) service.shutdown();
8  }
9  if(service != null) {
10    service.awaitTermination(1, TimeUnit.MINUTES);
11
12    // Check whether all tasks are finished
13    if(service.isTerminated()) System.out.println("Finished!");
14    else System.out.println("At least one task is still running");
15 }
```

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 synchronous, 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");
```

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();
```

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

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);
```

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

```
service.scheduleAtFixedRate(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.

Increasing Concurrency with Pools

A *thread pool* is a group of pre-instantiated reusable threads that are available to perform a set of arbitrary tasks.

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

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.

Writing Thread-Safe Code

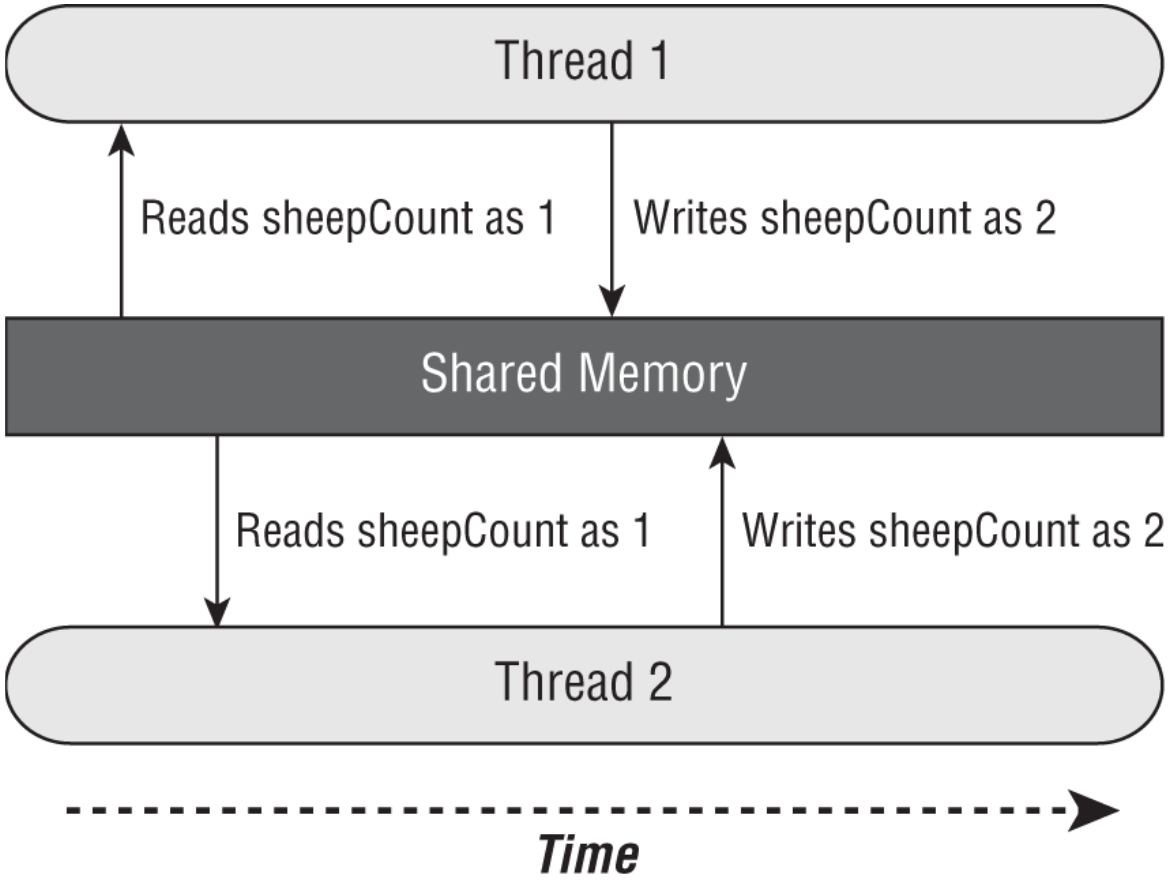
In this part of the chapter, we show how to use a variety of techniques to protect data including: atomic classes, `synchronized` blocks, the `Lock` framework, and cyclic barriers.

Understanding Thread-Safety

```
1  import java.util.concurrent.*;
2  public class SheepManager {
3     private int sheepCount = 0;
4     private void incrementAndReport() {
5        System.out.print((++sheepCount)+" ");
6     }
7     public static void main(String[] args) {
8        ExecutorService service = null;
9        try {
10          service = Executors.newFixedThreadPool(20);
11          SheepManager manager = new SheepManager();
12          for(int i = 0; i < 10; i++)
13             service.submit(() -> manager.incrementAndReport());
14       } finally {
15          if(service != null) service.shutdown();
16       }
17    }
18 }
```

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.

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*.

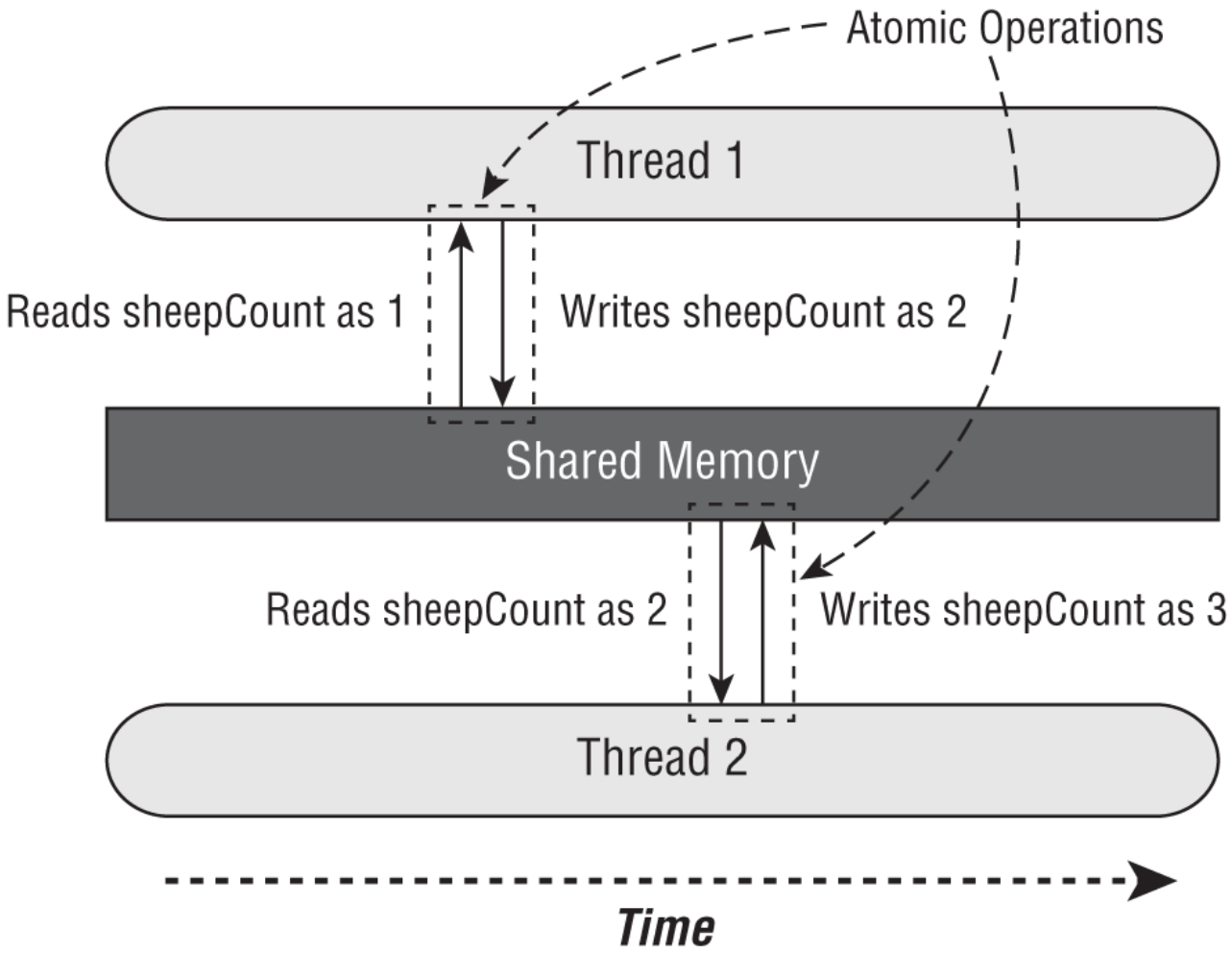


Protecting Data with Atomic Classes

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.

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.



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

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

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

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

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.

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
```

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)+" ");
   }
}
```

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)+" ");
}
```

Understanding the *Lock* Framework

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.

```
// 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();
}
```

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 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

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
```

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

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.

```
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");
}
```

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");
}
```

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.

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.

* `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*

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
```

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.

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.

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.

Using Concurrent Collections

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);
```

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

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

`ConcurrentHashMap` implements `Map` and `ConcurrentMap`

Understanding *SkipList* Collections

The `SkipList` classes, `ConcurrentSkipListSet` and `ConcurrentSkipListMap`, are concurrent versions of their sorted counterparts, `TreeSet` and `TreeMap`, respectively.

```
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

These classes copy all of their elements to a new underlying structure anytime an element is added, modified, or removed from the collection.

```
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 runtime.

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.

Understanding Blocking Queues

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.

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

```
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
```

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

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)`

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.

Identifying Threading Problems

Understanding Liveness

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.

Deadlock

Deadlock occurs when two or more threads are blocked forever, each waiting on the other.

```
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!
```

Starvation

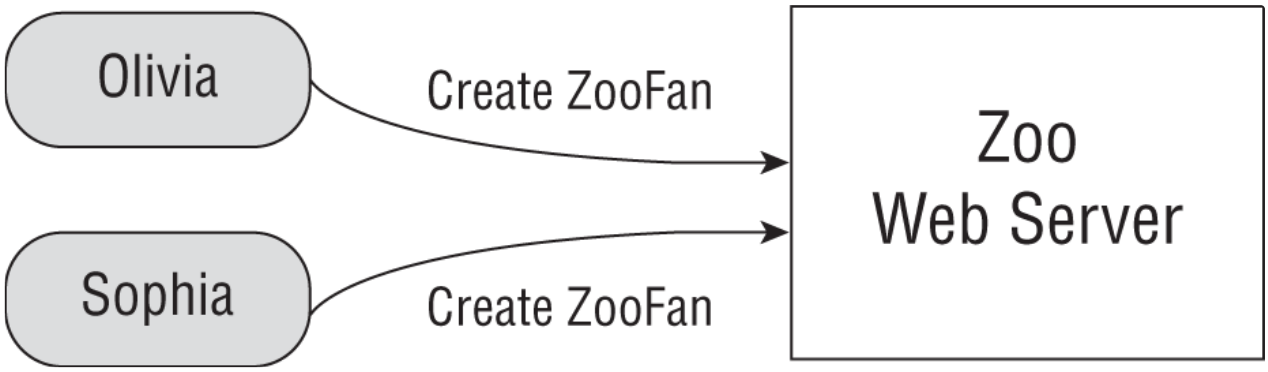
Starvation occurs when a single thread is perpetually denied access to a shared resource or lock. The thread is still active, but it is unable to complete its work as a result of other threads constantly taking the resource that they are trying to access.

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.

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.



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.

Working with Parallel Streams

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.

Creating Parallel Streams

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();
```

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

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();
```

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.

```
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
```

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
```

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
```

Processing Parallel Reductions

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.

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.

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();
```

Combining Results with *reduce()*

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.

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
```

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.

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*

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.ConcurrentHashMap
```

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

Finally, we can rewrite our `groupingBy()` example from Chapter 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.

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.