10

Threads

CERTIFICATION OBJECTIVES

- Create Worker Threads Using Runnable, Callable, and Use an ExecutorService to Concurrently Execute Tasks
- Identify Potential Threading Problems among Deadlock, Starvation, Livelock, and Race Conditions
- Use Synchronized keyword and java.util.concurrent.atomic Package to Control the Order of Thread Execution

✓ Two-Minute Drill

Q&A Self Test

CERTIFICATION OBJECTIVE

Defining, Instantiating, and Starting Threads (OCP Objective 10.1)

10.1 Create worker threads using Runnable, Callable, and use an ExecutorService to concurrently execute tasks.

Imagine a stockbroker application with a lot of complex capabilities. One of its functions is "download last stock option prices," another is "check prices for warnings," and a third time-consuming operation is "analyze historical data for company XYZ."

In a single-threaded runtime environment, these actions execute one after another. The next action can happen *only* when the previous one is finished. If a historical analysis takes half an hour, and the user selects to perform a download and check afterward, the warning may come too late to, say, buy or sell stock as a result.

We just imagined the sort of application that cries out for multithreading. Ideally, the download should happen in the background (that is, in another thread). That way, other processes could happen at the same time so that, for example, a warning could be communicated instantly. All the while, the user is interacting with other parts of the application. The analysis, too, could happen in a separate thread so the user can work in the

rest of the application while the results are being calculated.

So what exactly is a thread? In Java, "thread" means two different things:

- An instance of class `java.lang.Thread`
- A thread of execution

An instance of `Thread` is just…an object. Like any other object in Java, it has variables and methods, and it lives and dies on the heap. But a *thread of execution* is an individual process (a "lightweight" process) that has its own call stack. In Java, there is *one thread per call stack*—or, to think of it in reverse, *one call stack per thread.* Even if you don't create any new threads in your program, threads are back there running.

The `main()` method, which starts the whole ball rolling, runs in one thread, called (surprisingly) the *main* thread. If you looked at the main call stack (and you can any time you get a stack trace from something that happens after main begins, but not within another thread), you'd see that `main()` is the first method on the stack—the method at the bottom. But as soon as you create a *new* thread, a new stack materializes and methods called from *that* thread run in a call stack that's separate from the `main()` call stack. That second new call stack is said to run concurrently with the main thread, but we'll refine that notion as we go through this chapter.

You might find it confusing that we're talking about code running *concurrently*—what gives? The JVM, which gets its turn at the CPU by whatever scheduling mechanism the underlying OS uses, operates like a mini-OS and schedules *its* own threads, regardless of the underlying operating system. In some JVMs, the Java threads are actually mapped to native OS threads, but we won't discuss that here; native threads are not on the exam. Nor is it required to understand how threads behave in different JVM environments. In fact, the most important concept to understand from this entire chapter is this:

When it comes to threads, very little is guaranteed.

So be very cautious about interpreting the behavior you see on *one* machine as "the way threads work." The exam expects you to know what is and is not guaranteed behavior so that you can design your program in such a way that it will work, regardless of the underlying JVM. *That's part of the whole point of Java.*

Don't make the mistake of designing your program to be dependent on a particular implementation of the JVM. As you'll learn a little later, different JVMs can run threads in profoundly different ways. For example, one JVM might be sure that all threads get their turn, with a fairly even amount of time allocated for each thread in a nice, happy, round-

robin fashion. But in other JVMs, a thread might start running and then just hog the whole show, never stepping out so others can have a turn. If you test your application on the "nice turn-taking" JVM and you don't know what is and is not guaranteed in Java, then you might be in for a big shock when you run it under a JVM with a different thread-scheduling mechanism.

The thread questions are among the most difficult questions on the exam. In fact, for most people, they *are* the toughest questions on the exam, and with three objectives for threads, you'll be answering a *lot* of thread questions. If you're not already familiar with threads, you'll probably need to spend some time experimenting. Also, one final disclaimer: *This chapter makes almost no attempt to teach you how to design a good, safe multithreaded application. We only scratch the surface of that huge topic in this chapter!* You're here to learn the basics of threading and what you need to get through the thread questions on the exam. Before you can write decent multithreaded code, however, you really need to study more of the complexities and subtleties of multithreaded code.

Note: The topic of daemon threads is NOT on the exam. All of the threads discussed in this chapter are "user" threads. You and the operating system can create a second kind of thread called a daemon thread. The difference between these two types of threads (user and daemon) is that the JVM exits an application only when all user threads are complete—the JVM doesn't care about letting daemon threads complete, so once all user threads are complete, the JVM will shut down, regardless of the state of any daemon threads. Once again, this topic is NOT on the exam.

Making a Thread

A thread in Java begins as an instance of `java.lang.Thread`. You'll find methods in the `Thread` class for managing threads, including creating, starting, and pausing them. For the exam, you'll need to know, at a minimum, the following methods:

```
start()
yield()
sleep()
run()
```

The action happens in the `run()` method. Think of the code you want to execute in a separate thread as *the job to do.* In other words, you have some work that needs to be done —say, downloading stock prices in the background while other things are happening in the program—so what you really want is that *job* to be executed in its own thread. So, if the *work* you want done is the *job,* the one *doing* the work (actually executing the job code) is the *thread*. And the *job always starts from a* `run()` *method,* as follows:

```
public void run() {
  // your job code goes here
}
```

You always write the code that needs to be run in a separate thread in a `run()` method. The `run()` method will call other methods, of course, but the thread of execution—the new call stack—always begins by invoking `run()`. So where does the `run()` method go? In one of the two classes you can use to define your thread job.

You can define and instantiate a thread in one of two ways:

- Extend the `java.lang.Thread` class.
- Implement the `Runnable` interface.

You need to know about both for the exam, although in the real world, you're much more likely to implement `Runnable` than extend `Thread`. Extending the `Thread` class is the easiest, but it's usually not good OO practice. Why? Because subclassing should be reserved for specialized versions of more general superclasses. So the only time it really makes sense (from an OO perspective) to extend `Thread` is when you have a more specialized version of a `Thread` class. In other words, because *you have more specialized thread-specific behavior.* Chances are, though, that the thread work you want is really just a job to be done *by* a thread. In that case, you should design a class that implements the `Runnable` interface, which also leaves your class free to extend some *other* class.

Defining a Thread

To define a thread, you need a place to put your `run()` method, and as we just discussed, you can do that by extending the `Thread` class or by implementing the `Runnable` interface. We'll look at both in this section.

Extending java.lang.Thread

The simplest way to define code to run in a separate thread is to

- Extend the `java.lang.Thread` class.
- Override the `run()` method.

It looks like this:

```
class MyThread extends Thread {
  public void run() {
    System.out.println("Important job running in MyThread");
  }
}
```

The limitation with this approach (besides being a poor design choice in most cases) is that if you extend `Thread`, *you can't extend anything else.* And it's not as if you really need that inherited `Thread` class behavior; because in order to use a thread, you'll need to instantiate one anyway.

Keep in mind that you're free to overload the `run()` method in your `Thread` subclass:

```
class MyThread extends Thread {
  public void run() {
    System.out.println("Important job running in MyThread");
  }
  public void run(String s) {
    System.out.println("String in run is " + s);
  }
}
```

But know this: The overloaded `run(String s)` method will be ignored by the `Thread` class unless you call it yourself. The `Thread` class expects a `run()` method with no arguments, and it will execute this method for you in a separate call stack after the thread has been started. With a `run(String s)` method, the `Thread` class won't call the method for you, and even if you call the method directly yourself, execution won't happen in a new thread of execution with a separate call stack. It will just happen in the same call stack as the code that you made the call from, just like any other normal method call.

Implementing java.lang.Runnable

Implementing the `Runnable` interface gives you a way to extend any class you like but still define behavior that will be run by a separate thread. It looks like this:

```
class MyRunnable implements Runnable {
  public void run() {
    System.out.println("Important job running in MyRunnable");
  }
}
```

Notice that the `Runnable` interface is a functional interface, that is, an interface with one abstract method, `run()`. That means we can also write a `Runnable` like this:

```
Runnable r = () ->
   System.out.println("Important job running in a Runnable");
```

Regardless of which mechanism you choose, you've now got yourself some code that can be run by a thread of execution. Let's take a look at *instantiating* your thread-capable class, and then we'll figure out how to actually get the thing *running*.

Instantiating a Thread

Remember, every thread of execution begins as an instance of class `Thread`. Regardless of whether your `run()` method is in a `Thread` subclass or a `Runnable` implementation class, you still need a `Thread` object to do the work.

If you extended the `Thread` class, instantiation is dead simple (we'll look at some additional overloaded constructors in a moment):

```
MyThread t = new MyThread();
```

If you implement `Runnable`, instantiation is only slightly less simple. To have code run by a separate thread, *you still need a Thread instance.* But rather than combining both the *thread* and the *job* (the code in the `run()` method) into one class, you've split it into two classes—the `Thread` class for the *thread-specific* code and your `Runnable` implementation class for your *job-that-should-be-run-by-a-thread* code. (Another common way to think about this is that the `Thread` is the "worker," and the `Runnable` is the "job" to be done.)

First, you instantiate your `Runnable` class:

```
MyRunnable r = new MyRunnable();
```

Next, you get yourself an instance of `java.lang.Thread` (*somebody* has to run your job…), and you *give it your job!*

```
Thread t = new Thread(r);  // Pass your Runnable to the Thread
```

Or, if you want to use a lambda expression, you can eliminate `MyRunnable` and write:

```
Thread t = new Thread(
  () -> System.out.println("Important job running in a Runnable")
);
```

If you create a thread using the no-arg constructor, the thread will call its own `run()` method when it's time to start working. That's exactly what you want when you extend `Thread`, but when you use `Runnable`, you need to tell the new thread to use *your* `run()` method rather than its own. The `Runnable` you pass to the `Thread` constructor is called the *target* or the *target* `Runnable`.

You can pass a single `Runnable` instance to multiple `Thread` objects so that the same `Runnable` becomes the target of multiple threads, as follows:

```
public class TestThreads {
  public static void main (String [] args) {
    // MyRunnable r = new MyRunnable(); // OR get rid of MyRunnable and write:
    Runnable r = () ->
      System.out.println("Important job running in a Runnable");
      Thread foo = new Thread(r);
      Thread bar = new Thread(r);
      Thread bat = new Thread(r);
  }
}
```

Giving the same target to multiple threads means that several threads of execution will be running the very same job (and that the same job will be done multiple times).

The Thread class itself implements **`Runnable`**. *(After all, it has a* **`run()`** *method that we were overriding.) This means that you could pass a* **`Thread`** *to another Thread's constructor:*

```
Thread t = new Thread(new MyThread());
```

This is a bit silly, but it's legal. In this case, you really just need a ***Runnnable****, and creating a whole other* ***Thread*** *is overkill.*

Besides the no-arg constructor and the constructor that takes a `Runnable` (the target, i.e., the instance with the job to do), there are other overloaded constructors in class `Thread`. The constructors we care about are

- `Thread()`
- `Thread(Runnable target)`
- `Thread(Runnable target, String name)`
- `Thread(String name)`

You need to recognize all of them for the exam! A little later, we'll discuss some of the other constructors in the preceding list.

So now you've made yourself a `Thread` instance, and it knows which `run()` method to call. *But nothing is happening yet.* At this point, all we've got is a plain-old Java object of type `Thread`. *It is not yet a thread of execution*. To get an actual thread—a new call stack—we still have to *start* the thread.

When a thread has been instantiated but not started (in other words, the `start()` method has not been invoked on the `Thread` instance), the thread is said to be in the *new* state. At this stage, the thread is not yet considered *alive.* Once the `start()` method is called, the thread is considered *alive* (even though the `run()` method may not have actually started executing yet). A thread is considered *dead* (no longer *alive*) after the `run()` method completes. The `isAlive()` method is the best way to determine if a thread has been started but has not yet completed its `run()` method. (Note: The `getState()` method is very useful for debugging, but you don't have to know it for the exam.)

Starting a Thread

You've created a `Thread` object and it knows its target (either the passed-in `Runnable` or itself, if you extended class `Thread`). Now it's time to get the whole thread thing happening—to launch a new call stack. It's so simple; it hardly deserves its own subheading:

```
t.start();
```

Prior to calling `start()` on a `Thread` instance, the thread (when we use lowercase `t`, we're referring to the *thread of execution* rather than the `Thread` class) is said to be in the *new* state, as we said. The new state means you have a `Thread` *object* but you don't yet have a *true thread*. So what happens after you call `start()`? The good stuff:

- A new thread of execution starts (with a new call stack).

- The thread moves from the *new* state to the *runnable* state.
- When the thread gets a chance to execute, its target `run()` method will run.

Be *sure* you remember the following: You start a *Thread,* not a *Runnable.* You call `start()` on a `Thread` instance, not on a `Runnable` instance. The following example demonstrates what we've covered so far—defining, instantiating, and starting a thread:

```
public class TestThreads {
  public static void main (String [] args) {
    Runnable r = () -> {
      for (int x = 1; x < 6; x++) {
        System.out.println("Runnable running " + x);
      }
    };
    Thread t = new Thread(r);
    t.start();
  }
}
```

Running the preceding code prints out exactly what you'd expect:

```
% java TestThreads
Runnable running 1
Runnable running 2
Runnable running 3
Runnable running 4
Runnable running 5
```

(If this isn't what you expected, go back and reread everything in this objective.)

There's nothing special about the **`run()`** *method as far as Java is concerned. Like* **`main()`**, *it just happens to be the name (and signature) of the method that the new thread knows to invoke. So if you see code that calls the* **`run()`** *method on a Runnable (or even on a* **`Thread`** *instance), that's perfectly legal. But it doesn't mean the* **`run()`** *method will run in a separate thread! Calling a* **`run()`** *method directly just means you're invoking a method from whatever thread is currently executing, and the* **`run()`** *method goes onto the current call stack rather than at the beginning of a new call stack. The following code does not start a new thread of execution:*

```
Thread t = new Thread();
t.run();              // Legal, but does not start a new thread
```

So what happens if we start multiple threads? We'll run a simple example in a moment, but first we need to know how to print out which thread is executing. We can use the `getName()` method of class `Thread` and have each `Runnable` print out the name of the thread executing that `Runnable` object's `run()` method. The following example instantiates a thread and gives it a name, and then the name is printed out from the `run()` method:

```
class NameRunnable implements Runnable {
  public void run() {
    System.out.println("NameRunnable running");
    System.out.println("Run by "
      + Thread.currentThread().getName());
  }
}
public class NameThread {
  public static void main (String [] args) {
    NameRunnable nr = new NameRunnable();
    Thread t = new Thread(nr);
```

```
        t.setName("Fred");
        t.start();
    }
}
```

Running this code produces the following extra-special output:

```
% java NameThread
NameRunnable running
Run by Fred
```

To get the name of a thread, you call—who would have guessed—`getName()` on the `Thread` instance. But the target `Runnable` instance doesn't even *have* a reference to the `Thread` instance, so we first invoked the `static Thread.currentThread()` method, which returns a reference to the currently executing thread, and then we invoked `getName()` on that returned reference.

Even if you don't explicitly name a thread, it still has a name. Let's look at the previous code, commenting out the statement that sets the thread's name:

```
public class NameThread {
    public static void main (String [] args) {
        NameRunnable nr = new NameRunnable();
        Thread t = new Thread(nr);
        // t.setName("Fred");
        t.start();
    }
}
```

Running the preceding code now gives us

```
% java NameThread
NameRunnable running
Run by Thread-0
```

And since we're getting the current thread by using the static `Thread.`*`currentThread`*`()` method, we can even get the name of the thread running our main code:

```
public class NameThreadTwo {
  public static void main (String [] args) {
    System.out.println("thread is "
      + Thread.currentThread().getName());
  }
}
```

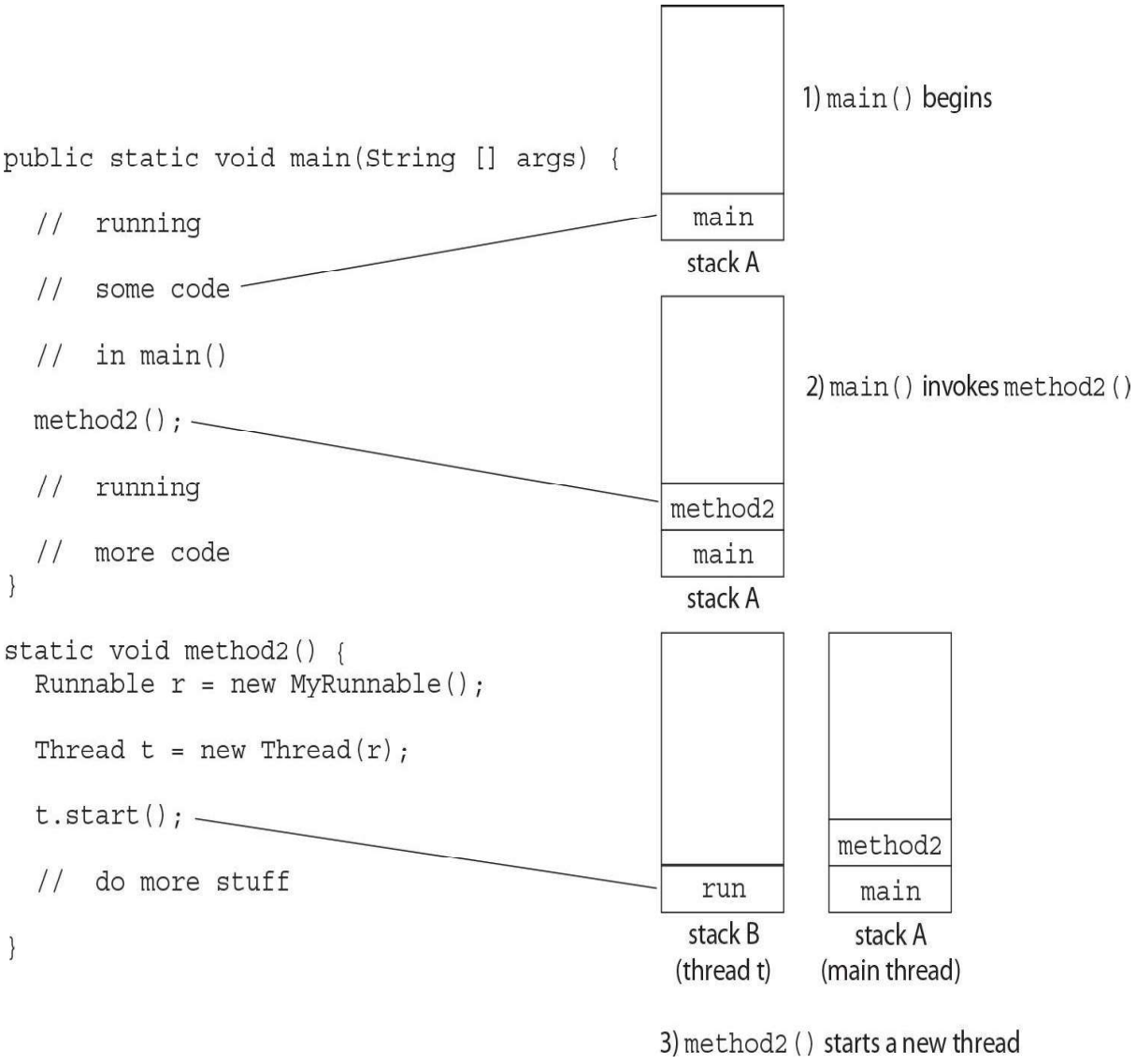
which prints out

```
% java NameThreadTwo
thread is main
```

That's right, the main thread already has a name—*main*. (Once again, what are the odds?) Figure 10-1 shows the process of starting a thread.

FIGURE 10-1

Starting a thread



Starting and Running Multiple Threads

Enough playing around here; let's actually get multiple threads going (more than two, that is). We already had two threads, because the `main()` method starts in a thread of its own, and then `t.start()` started a *second* thread. Now we'll do more. The following code creates a single `Runnable` instance and three `Thread` instances. All three `Thread` instances get the same `Runnable` instance, and each thread is given a unique name. Finally, all three threads are started by invoking `start()` on the `Thread` instances. And just to hammer in one more time how you can use a lambda expression in place of an explicit `Runnable` class, we'll eliminate `NameRunnable`, and replace it with a lambda.

```
// No need for NameRunnable if we use lambdas…
public class ManyNames {
   public static void main(String [] args) {

    // Make one Runnable
    Runnable nr = () -> {
      for (int x = 1; x <= 3; x++) {
        System.out.println("Run by " +
          Thread.currentThread().getName() + ", x is " + x);
      }
    };
    Thread one = new Thread(nr);
    Thread two = new Thread(nr);
    Thread three = new Thread(nr);

    one.setName("Fred");
    two.setName("Lucy");
    three.setName("Ricky");
    one.start();
    two.start();
    three.start();
  }
}
```

Running this code might produce the following:

```
% java ManyNames
Run by Fred, x is 1
Run by Fred, x is 2
Run by Fred, x is 3
Run by Lucy, x is 1
Run by Lucy, x is 2
Run by Lucy, x is 3
Run by Ricky, x is 1
Run by Ricky, x is 2
Run by Ricky, x is 3
```

Well, at least that's what it printed when we ran it—this time, on our machine. But the behavior you see here is not guaranteed. This is so crucial that you need to stop right now, take a deep breath, and repeat after me, "The behavior is not guaranteed." You need to know, for your future as a Java programmer as well as for the exam, that there is nothing in the Java specification that says threads will start running in the order in which they were started (in other words, the order in which `start()` was invoked on each thread). And there is no guarantee that once a thread starts executing, it will keep executing until it's done. Or that a loop will complete before another thread begins. No siree, Bob.

Nothing is guaranteed in the preceding code except this:

Each thread will start, and each thread will run to completion.

Within each thread, things will happen in a predictable order. But the actions of different threads can mix in unpredictable ways. If you run the program multiple times or on multiple machines, you may see different output. Even if you don't see different output, you need to realize that the behavior you see is not guaranteed. Sometimes a little change in the way the program is run will cause a difference to emerge. Just for fun we bumped up the loop code so that each `run()` method ran the `for` loop 400 times rather than 3, and eventually we did start to see some wobbling:

```
Runnable nr = () -> {
  for (int x = 1; x <= 400; x++) {
    System.out.println("Run by " +
      Thread.currentThread().getName() + ", x is " + x);
  }
};
```

Running the preceding code, with each thread executing its run loop 400 times, started out fine but then became nonlinear. Here's just a snippet from the command-line output of running that code. To make it easier to distinguish each thread, we put Fred's output in italics and Lucy's in bold and left Ricky's alone:

```
Run by Fred, x is 345
Run by Ricky, x is 313
Run by Lucy, x is 341
Run by Ricky, x is 314
Run by Lucy, x is 342
Run by Ricky, x is 315
Run by Fred, x is 346
Run by Lucy, x is 343
Run by Fred, x is 347
Run by Lucy, x is 344
```

…it continues on…

Notice that there's not really any clear pattern here. If we look at only the output from

Fred, we see the numbers increasing one at a time, as expected:

```
Run by Fred, x is 345
Run by Fred, x is 346
Run by Fred, x is 347
```

And similarly, if we look only at the output from Lucy or Ricky—each one individually is behaving in a nice, orderly manner. But together—chaos! In the previous fragment we see Fred, then Lucy, then Ricky (in the same order we originally started the threads), but then Lucy butts in when it was Fred's turn. What nerve! And then Ricky and Lucy trade back and forth for a while until finally Fred gets another chance. They jump around like this for a while after this. Eventually (after the part shown earlier), Fred finishes, then Ricky, and finally Lucy finishes with a long sequence of output. So even though Ricky was started third, he actually completed second. And if we run it again, we'll get a different result. Why? Because it's up to the scheduler, and we don't control the scheduler! Which brings up another key point to remember: Just because a series of threads are started in a particular order doesn't mean they'll run in that order. For any group of started threads, order is not guaranteed by the scheduler. And duration is not guaranteed. You don't know, for example, if one thread will run to completion before the others have a chance to get in, or whether they'll all take turns nicely, or whether they'll do a combination of both. There is a way, however, to start a thread but tell it not to run until some other thread has finished. You can do this with the `join()` method, which we'll look at a little later.

A thread is done being a thread when its target ***`run()`*** *method completes.*

When a thread completes its `run()` method, the thread ceases to be a thread of execution. The stack for that thread dissolves, and the thread is considered dead. (Technically, the API calls a dead thread "terminated," but we'll use "dead" in this chapter.) Not dead and gone, however—just dead. It's still a `Thread` *object,* just not a *thread of execution.* So if you've got a reference to a `Thread` instance, then even when that `Thread` instance is no longer a thread of execution, you can still call methods on the `Thread` instance, just like any other Java object. What you can't do, though, is call `start()` again.

Once a thread has been started, it can never be started again.

If you have a reference to a `Thread` and you call `start()`, it's started. If you call `start()` a second time, it will cause an exception (an `IllegalThreadStateException`, which is a kind of `RuntimeException`, but you don't need to worry about the exact type). This happens whether or not the `run()` method has completed from the first `start()` call. Only a new thread can be started, and then only once. A runnable thread or a dead thread cannot be restarted.

So far, we've seen three thread states: *new, runnable,* and *dead.* We'll look at more thread states before we're done with this chapter.

In addition to using **`setName()`** *and* **`getName`** *to identify threads, you might see* **`getId()`**. *The* **`getId()`** *method returns a positive, unique* **`long`** *number, and that number will be that thread's only ID number for the thread's entire life.*

The Thread Scheduler

The thread scheduler is the part of the JVM (although most JVMs map Java threads directly to native threads on the underlying OS) that decides which thread should run at any given moment and also takes threads *out* of the run state. Assuming a single processor machine, only one thread can actually *run* at a time. Only one stack can ever be executing at one time. And it's the thread scheduler that decides *which* thread—of all that are eligible—will actually *run*. When we say *eligible,* we really mean *in the runnable state*.

Any thread in the *runnable* state can be chosen by the scheduler to be the one and only running thread. If a thread is not in a runnable state, then it cannot be chosen to be the *currently running* thread. And just so we're clear about how little is guaranteed here:

The order in which runnable threads are chosen to run is not guaranteed.

Although *queue* behavior is typical, it isn't guaranteed. Queue behavior means that when a thread has finished with its "turn," it moves to the end of the line of the runnable pool and waits until it eventually gets to the front of the line, where it can be chosen again. In fact, we call it a runnable *pool,* rather than a runnable *queue,* to help reinforce the fact that threads aren't all lined up in some guaranteed order.

Although we don't *control* the thread scheduler (we can't, for example, tell a specific thread to run), we can sometimes influence it. The following methods give us some tools for *influencing* the scheduler. Just don't ever mistake influence for control.

Expect to see exam questions that look for your understanding of what is and is not guaranteed! You must be able to look at thread code and determine whether the output is guaranteed to run in a particular way or is unpredictable.

Methods from the java.lang.Thread Class Some of the methods that can help us influence thread scheduling are as follows:

```
public static void sleep(long millis) throws InterruptedException
public static void yield()
public final void join() throws InterruptedException
public final void setPriority(int newPriority)
```

Note that both `sleep()` and `join()` have overloaded versions not shown here.

Methods from the java.lang.Object Class Every class in Java inherits the following three thread-related methods:

```
public final void wait() throws InterruptedException
public final void notify()
public final void notifyAll()
```

The `wait()` method has three overloaded versions (including the one listed here).

We'll look at the behavior of each of these methods in this chapter. First, though, we're going to look at the different states a thread can be in.

Thread States and Transitions

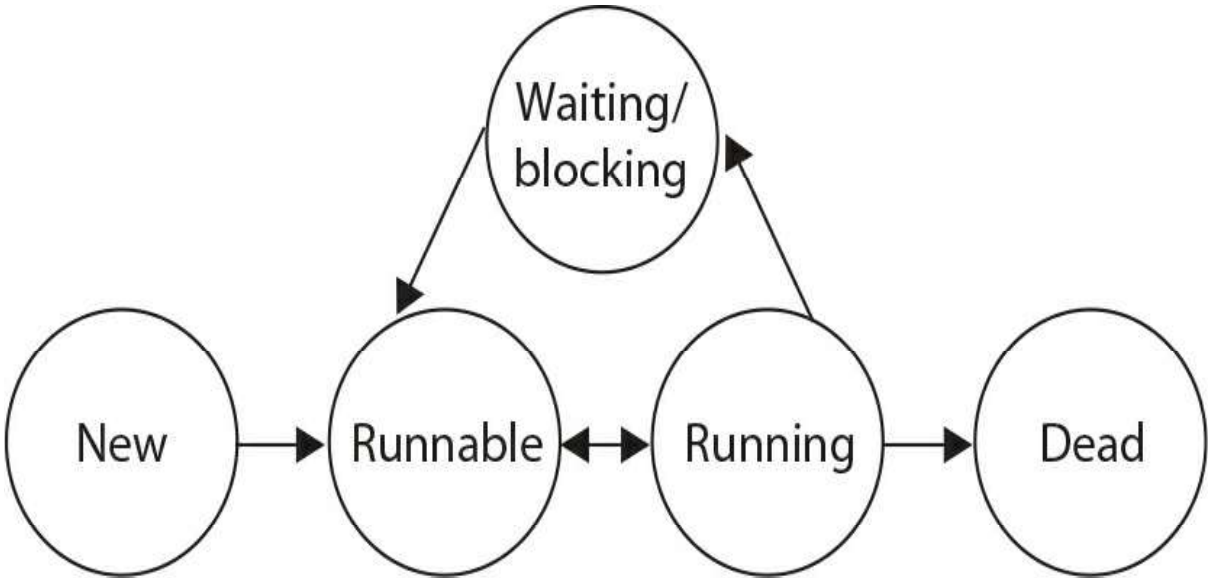
We've already seen three thread states—*new, runnable,* and *dead*—but wait! There's more! The thread scheduler's job is to move threads in and out of the *running* state. While the thread scheduler can move a thread from the running state back to runnable, other factors can cause a thread to move out of running, but *not* back to runnable. One of these is when the thread's `run()` method completes, in which case, the thread moves from the running state directly to the dead state. Next, we'll look at some of the other ways in which a thread can leave the running state and where the thread goes.

Thread States

A thread can be only in one of five states (see Figure 10-2):

FIGURE 10-2

Transitioning between thread states



- New This is the state the thread is in after the `Thread` instance has been created but the `start()` method has not been invoked on the thread. It is a live `Thread` object, but not yet a thread of execution. At this point, the thread is considered *not alive*.
- Runnable This is the state a thread is in when it's eligible to run but the scheduler has not selected it to be the running thread. A thread first enters the runnable state when the `start()` method is invoked, but a thread can also return to the runnable state after either running or coming back from a blocked, waiting, or sleeping state. When the thread is in the runnable state, it is considered *alive*.
- Running This is it. The "big time." Where the action is. This is the state a thread is in when the thread scheduler selects it from the runnable pool to be the currently executing process. A thread can transition out of a running state for several reasons, including because "the thread scheduler felt like it." We'll look at those other reasons shortly. Note that in Figure 10-2, there are several ways to get to the runnable state, but only *one* way to get to the running state: the scheduler chooses a thread from the runnable pool.
- Waiting/blocked/sleeping This is the state a thread is in when it's not eligible to run. Okay, so this is really three states combined into one, but they all have one thing in common: the thread is still alive but is currently not eligible to run. In other words, it is not *runnable,* but it might *return* to a runnable state later if a particular event occurs. A thread may be *blocked* because it's waiting for a resource (like I/O or an object's lock), in which case the event that sends it back to runnable is the availability of the resource—for example, if data comes in through the input stream the thread code is reading from or if the object's lock suddenly becomes available. A thread may be *sleeping* because the thread's run code *tells* it to sleep for some period of time, in which case, the event that sends it back to runnable causes it to wake up because its sleep time has expired. Or the thread may be *waiting*

because the thread's run code *causes* it to wait. In that case, an event occurs, causing another thread to be sent a notifcation that it may no longer be necessary to wait. Then the waiting thread will become runnable again. The important point is that one thread does not *tell* another thread to block. Some methods may *look* like they tell another thread to block, but they don't. If you have a reference `t` to another thread, you can write something like this:

```
t.sleep();    or    t.yield();
```

But those are actually static methods of the `Thread` class—*they don't affect the instance* `t`; instead, they are defined to always affect the thread that's currently executing. (This is a good example of why it's a bad idea to use an instance variable to access a `static` method—it's misleading. There *is* a method, `suspend()`, in the `Thread` class that lets one thread tell another to suspend, but the `suspend()` method has been deprecated and won't be on the exam [nor will its counterpart `resume()`].) There is also a `stop()` method, but it, too, has been deprecated and we won't even go there. Both `suspend()` and `stop()` turned out to be very dangerous, so you shouldn't use them, and again, because they're deprecated, they won't appear on the exam. Don't study 'em; don't use 'em. Note also that a thread in a blocked state is still considered *alive*.

- Dead A thread is considered dead when its `run()` method completes. It may still be a viable `Thread` object, but it is no longer a separate thread of execution. Once a thread is dead, it can never be brought back to life! (The whole "I see dead threads" thing.) If you invoke `start()` on a dead `Thread` instance, you'll get an exception at runtime. And it probably doesn't take a rocket scientist to tell you that if a thread is dead, it is no longer considered *alive*.

Preventing Thread Execution

A thread that's been stopped usually means a thread that's moved to the dead state. But you also need to be able to recognize when a thread will get kicked out of running but *not* be sent back to either runnable or dead.

For the purpose of the exam, we aren't concerned with a thread blocking on I/O (say, waiting for something to arrive from an input stream from the server). We *are* concerned with the following:

- Sleeping
- Waiting
- Blocked because it needs an object's lock

Sleeping

The `sleep()` method is a `static` method of class `Thread`. You use it in your code to

"slow a thread down" by forcing it to go into a sleep mode before coming back to runnable (where it still has to beg to be the currently running thread). When a thread sleeps, it drifts off somewhere and doesn't return to runnable until it wakes up.

So why would you want a thread to sleep? Well, you might think the thread is moving too quickly through its code. Or you might need to force your threads to take turns, since reasonable turn-taking isn't guaranteed in the Java specification. Or imagine a thread that runs in a loop, downloading the latest stock prices and analyzing them. Downloading prices one after another would be a waste of time, as most would be quite similar—and even more important, it would be an incredible waste of precious bandwidth. The simplest way to solve this is to cause a thread to pause (sleep) for five minutes after each download.

You do this by invoking the static `Thread.sleep()` method, giving it a time in milliseconds as follows:

```
try {
  Thread.sleep(5*60*1000);  // Sleep for 5 minutes
} catch (InterruptedException ex) { }
```

Notice that the `sleep()` method can throw a checked `InterruptedException` (you'll usually know if that is a possibility because another thread has to explicitly do the interrupting), so you must acknowledge the exception with a handle or declare. Typically, you wrap calls to `sleep()` in a `try/catch`, as in the preceding code.

Let's modify our Fred, Lucy, Ricky code by using `sleep()` to *try* to force the threads to alternate rather than letting one thread dominate for any period of time. Where do you think the call to the `sleep()` method should go?

```
class NameRunnable implements Runnable {
  public void run() {
    for (int x = 1; x < 4; x++) {
      System.out.println("Run by "
        + Thread.currentThread().getName());
      try {
        Thread.sleep(1000);
      } catch (InterruptedException ex) { }
    }
  }
}
public class ManyNames {
  public static void main (String [] args) {

    // Make one Runnable
    NameRunnable nr = new NameRunnable();

    Thread one = new Thread(nr);
    one.setName("Fred");
    Thread two = new Thread(nr);
    two.setName("Lucy");
    Thread three = new Thread(nr);
```

```
        three.setName("Ricky");

        one.start();
        two.start();
        three.start();
    }
}
```

Running this code shows Fred, Lucy, and Ricky alternating nicely:

```
% java ManyNames
Run by Fred
Run by Lucy
Run by Ricky
Run by Fred
Run by Lucy
Run by Ricky
Run by Fred
Run by Lucy
Run by Ricky
```

Just keep in mind that the behavior in the preceding output is still not guaranteed. You can't be certain how long a thread will actually run *before* it gets put to sleep, so you can't know with certainty that only one of the three threads will be in the runnable state when the running thread goes to sleep. In other words, if two threads are awake and in the runnable pool, you can't know with certainty that the least recently used thread will be the one selected to run. *Still, using sleep() is the best way to help all threads get a chance to run!* Or at least to guarantee that one thread doesn't get in and stay until it's done. When a thread encounters a sleep call, it *must* go to sleep for *at least* the specified number of milliseconds (unless it is interrupted before its wake-up time, in which case, it immediately throws the `InterruptedException`).

Just because a thread's **`sleep()`** *expires and it wakes up does not mean it will return to running! Remember, when a thread wakes up, it simply goes back to the runnable state. So the time specified in* **`sleep()`** *is the minimum duration in which the thread won't run, but it is not the exact duration in which the thread won't run. So you can't, for example, rely on the* **`sleep()`** *method to give you a perfectly accurate timer. Although in many applications using* **`sleep()`** *as a timer is certainly good enough, you must know that a* **`sleep()`** *time is not a guarantee that the thread will start running again as soon as the time expires and the thread wakes.*

Remember that `sleep()` is a static method, so don't be fooled into thinking that one thread can put another thread to sleep. You can put `sleep()` code anywhere since *all* code is being run by *some* thread. When the executing code (meaning the currently running thread's code) hits a `sleep()` call, it puts the currently running thread to sleep.

EXERCISE 10-1

Creating a Thread and Putting It to Sleep

In this exercise, we will create a simple counting thread. It will count to 100, pausing one second between each number. Also, in keeping with the counting theme, it will output a string every ten numbers.

1. Create a class and extend the `Thread` class. As an option, you can implement the `Runnable` interface.
2. Override the `run()` method of `Thread`. This is where the code will go that will output the numbers.
3. Create a `for` loop that will loop 100 times. Use the modulus operation to check whether there are any remainder numbers when divided by 10.
4. Use the static method `Thread.sleep()` to pause. (Remember, the one-arg version of `sleep()` specifies the amount of time of sleep in milliseconds.)

Thread Priorities and yield( )

To understand `yield()`, you must understand the concept of thread *priorities.* Threads always run with some priority, usually represented as a number between 1 and 10 (although in some cases, the range is less than 10). The scheduler in most JVMs uses preemptive,

priority-based scheduling (which implies some sort of time slicing). *This does not mean that all JVMs use time slicing.* The JVM specification does not require a VM to implement a time-slicing scheduler, where each thread is allocated a fair amount of time and then sent back to runnable to give another thread a chance. Although many JVMs do use time slicing, some may use a scheduler that lets one thread stay running until the thread completes its `run()` method.

In most JVMs, however, the scheduler does use thread priorities in one important way: If a thread enters the runnable state and it has a higher priority than any of the threads in the pool and a higher priority than the currently running thread, *the lower-priority running thread usually will be bumped back to runnable and the highest-priority thread will be chosen to run.* In other words, at any given time, the currently running thread usually will not have a priority that is lower than any of the threads in the pool. *In most cases, the running thread will be of equal or greater priority than the highest-priority threads in the pool.* This is as close to a guarantee about scheduling as you'll get from the JVM specification, so you must never rely on thread priorities to guarantee the correct behavior of your program.

Don't rely on thread priorities when designing your multithreaded application. Because thread-scheduling priority behavior is not guaranteed, it's better to avoid modifying thread priorities. Usually, default priority will be fine.

What is also *not* guaranteed is the behavior when threads in the pool are of equal priority or when the currently running thread has the same priority as threads in the pool. All priorities being equal, a JVM implementation of the scheduler is free to do just about anything it likes. That means a scheduler might do one of the following (among other things):

- Pick a thread to run, and run it there until it blocks or completes.
- Time-slice the threads in the pool to give everyone an equal opportunity to run.

Setting a Thread's Priority

A thread gets a default priority that is *the priority of the thread of execution that creates it.* For example, in the code

```
public class TestThreads {
  public static void main (String [] args) {
    MyThread t = new MyThread();
  }
}
```

the thread referenced by `t` will have the same priority as the *main* thread because the main thread is executing the code that creates the `MyThread` instance.

You can also set a thread's priority directly by calling the `setPriority()` method on a `Thread` instance, as follows:

```
FooRunnable r = new FooRunnable();
Thread t = new Thread(r);
t.setPriority(8);
t.start();
```

Priorities are set using a positive integer, usually between 1 and 10, and the JVM will never change a thread's priority. However, values 1 through 10 are not guaranteed. Some JVMs might not recognize 10 distinct values. Such a JVM might merge values from 1 to 10 down to maybe values from 1 to 5, so if you have, say, 10 threads, each with a different priority, and the current application is running in a JVM that allocates a range of only 5 priorities, then 2 or more threads might be mapped to one priority.

Although *the default priority is 5*, the `Thread` class has the three following constants (`static final` variables) that define the range of thread priorities:

```
Thread.MIN_PRIORITY   (1)
Thread.NORM_PRIORITY   (5)
Thread.MAX_PRIORITY   (10)
```

The yield( ) Method

So what does the `static Thread.yield()` have to do with all this? Not that much, in practice. What `yield()` is *supposed* to do is make the currently running thread head back to runnable to allow other threads of the same priority to get their turn. So the intention is to use `yield()` to promote graceful turn-taking among equal-priority threads. In reality, though, the `yield()` method isn't guaranteed to do what it claims, and even if `yield()` does cause a thread to step out of running and back to runnable, *there's no guarantee the yielding thread won't just be chosen again over all the others!* So while `yield()` might—and

often does—make a running thread give up its slot to another runnable thread of the same priority, there's no guarantee.

A `yield()` won't ever cause a thread to go to the waiting/sleeping/blocking state. At most, a `yield()` will cause a thread to go from running to runnable, but again, it might have no effect at all.

The join( ) Method

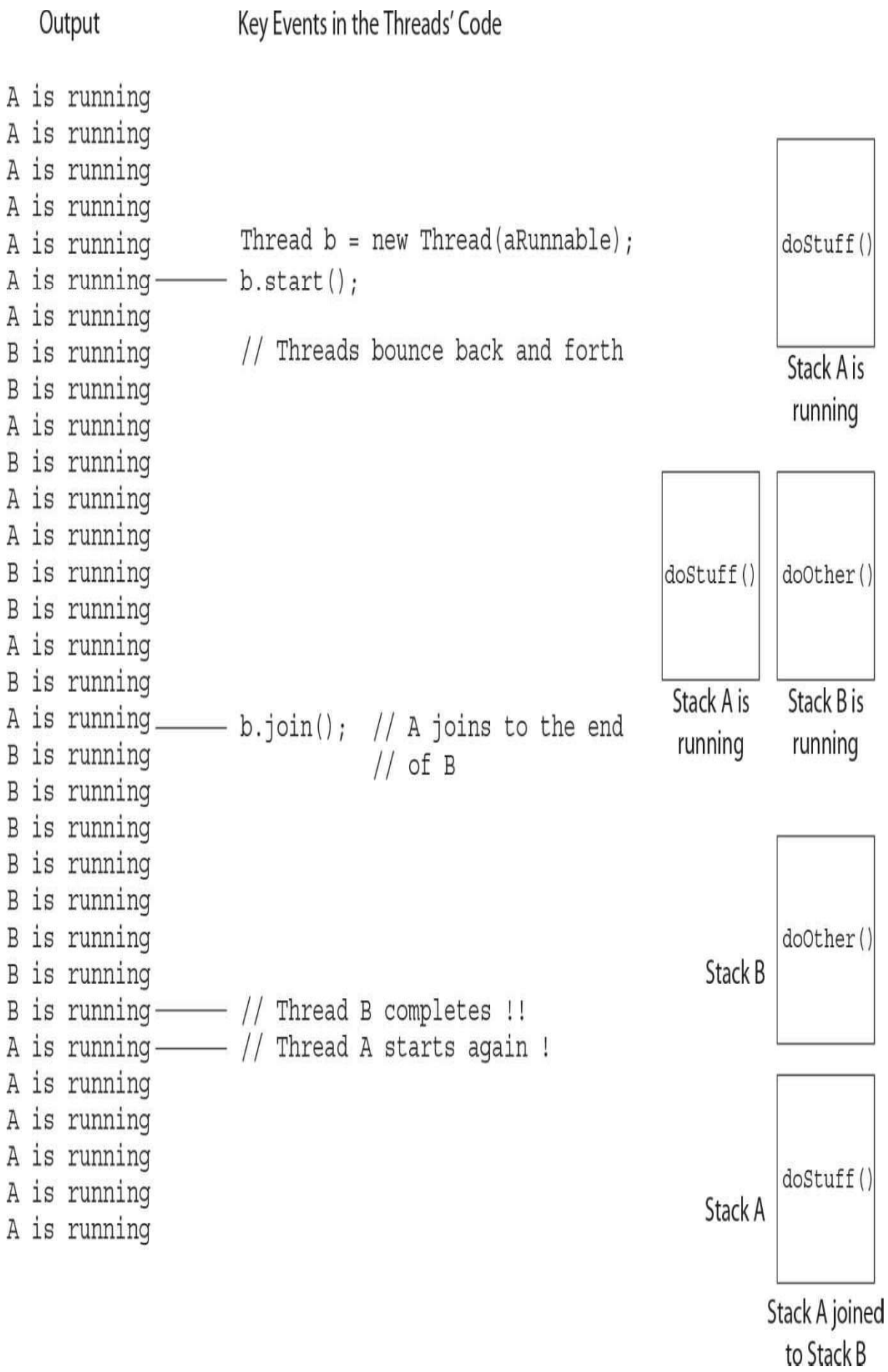
The non-`static join()` method of class `Thread` lets one thread "join onto the end" of another thread. If you have a thread B that can't do its work until another thread A has completed *its* work, then you want thread B to "join" thread A. This means that thread B will not become runnable until A has finished (and entered the dead state).

```
Thread t = new Thread();
t.start();
t.join();
```

The preceding code takes the currently running thread (if this were in the `main()` method, then that would be the main thread) and *joins* it to the end of the thread referenced by `t`. This blocks the current thread from becoming runnable until after the thread referenced by `t` is no longer alive. In other words, the code `t.join()` means "Join me (the current thread) to the end of `t`, so that `t` must finish before I (the current thread) can run again." You can also call one of the overloaded versions of `join()` that takes a timeout duration so that you're saying, "Wait until thread `t` is done, but if it takes longer than 5,000 milliseconds, then stop waiting and become runnable anyway." Figure 10-3 shows the effect of the `join()` method.

FIGURE 10-3

The `join()` method


Output
Key Events in the Threads' Code
A is running
A is running
A is running
A is running
A is running
A is running
A is running
B is running
B is running
A is running
B is running
A is running
A is running
B is running
B is running
A is running
B is running
A is running
B is running
B is running
B is running
B is running
B is running
B is running
B is running
B is running
B is running
A is running
A is running
A is running
A is running
A is running
Thread b = new Thread(aRunnable);
b.start();
// Threads bounce back and forth
b.join(); // A joins to the end
// of B
// Thread B completes !!
// Thread A starts again !
doStuff()
Stack A is running
doStuff()
doOther()
Stack A is running
Stack B is running
Stack B
doOther()
Stack A
doStuff()
Stack A joined to Stack B

So far, we've looked at three ways a running thread could leave the running state:

- A call to **`sleep()`** Guaranteed to cause the current thread to stop executing for at least the specified sleep duration (although it might be *interrupted* before its specified time).
- A call to **`yield()`** Not guaranteed to do much of anything, although typically, it will cause the currently running thread to move back to runnable so that a thread of the same priority can have a chance.
- A call to **`join()`** Guaranteed to cause the current thread to stop executing until the thread it joins with (in other words, the thread it calls `join()` on) completes, or if the thread it's trying to join with is not alive, the current thread won't need to back out.

Besides those three, we also have the following scenarios in which a thread might leave the running state:

- The thread's `run()` method completes. Duh.
- A call to `wait()` on an object (we don't call `wait()` on a *thread,* as we'll see in a moment).
- A thread can't acquire the *lock* on the object whose method code it's attempting to run.
- The thread scheduler can decide to move the current thread from running to runnable in order to give another thread a chance to run. No reason is needed—the thread scheduler can trade threads in and out whenever it likes.

CERTIFICATION OBJECTIVE

Synchronizing Code, Thread Problems (OCP Objectives 10.2 and 10.3)

10.2 Identify potential threading problems among deadlock, starvation, livelock, and race conditions.
10.3 Use synchronized keyword and java.util.concurrent.atomic package to control the order of thread execution.

Can you imagine the havoc that can occur when two different threads have access to a single instance of a class, and both threads invoke methods on that object…and those methods modify the state of the object? In other words, what might happen if *two* different threads call, say, a setter method on a *single* object? A scenario like that might corrupt an object's state by changing its instance variable values in an inconsistent way, and if that object's state is data shared by other parts of the program, well, it's too scary to even

visualize.

But just because we enjoy horror, let's look at an example of what might happen. The following code demonstrates what happens when two different threads are accessing the same account data. Imagine that two people each have a checkbook for a single checking account (or two people each have ATM cards, but both cards are linked to only one account).

In this example, we have a class called Account that represents a bank account. To keep the code short, this account starts with a balance of 50 and can be used only for withdrawals. The withdrawal will be accepted even if there isn't enough money in the account to cover it. The account simply reduces the balance by the amount you want to withdraw:

```
class Account {
  private int balance = 50;
  public int getBalance() {
    return balance;
  }
  public void withdraw(int amount) {
    balance = balance - amount;
  }
}
```

Now here's where it starts to get fun. Imagine a couple, Fred and Lucy, who both have access to the account and want to make withdrawals. But they don't want the account to ever be overdrawn, so just before one of them makes a withdrawal, he or she will first check the balance to be certain there's enough to 670cover the withdrawal. Also, withdrawals are always limited to an amount of 10, so there must be at least 10 in the account balance in order to make a withdrawal. Sounds reasonable. But that's a two-step process:

1. Check the balance.
2. If there's enough in the account (in this example, at least 10), make the withdrawal.

What happens if something separates step 1 from step 2? For example, imagine what would happen if Lucy checks the balance and sees there's just exactly enough in the account, 10. *But before she makes the withdrawal, Fred checks the balance and also sees that there's enough for his withdrawal.* Since Lucy has verified the balance but not yet made her withdrawal, Fred is seeing "bad data." He is seeing the account balance *before* Lucy actually

debits the account, but at this point, that debit is certain to occur. Now both Lucy and Fred believe there's enough to make their withdrawals. Now imagine that Lucy makes *her* withdrawal, so there isn't enough in the account for Fred's withdrawal, but he thinks there is because when he checked, there was enough! Yikes. In a minute, we'll see the actual banking code, with Fred and Lucy, represented by two threads, each acting on the same `Runnable`, and that `Runnable` holds a reference to the one and only account instance—so, two threads, one account.

The logic in our code example is as follows:

1. The `Runnable` object holds a reference to a single account.
2. Two threads are started, representing Lucy and Fred, and each thread is given a reference to the same `Runnable` (which holds a reference to the actual account).
3. The initial balance on the account is 50, and each withdrawal is exactly 10.
4. In the `run()` method, we loop five times, and in each loop we
   - Make a withdrawal (if there's enough in the account).
   - Print a statement *if the account is overdrawn* (which it should never be since we check the balance *before* making a withdrawal).
5. The `makeWithdrawal()` method in the test class (representing the behavior of Fred or Lucy) will do the following:
   - Check the balance to see if there's enough for the withdrawal.
   - If there is enough, print out the name of the one making the withdrawal.
   - Go to sleep for 500 milliseconds—just long enough to give the other partner a chance to get in before you actually *make* the withdrawal.
   - Upon waking up, complete the withdrawal and print that fact.
   - If there wasn't enough in the first place, print a statement showing who you are and the fact that there wasn't enough.

So what we're really trying to discover is if the following is possible: for one partner to check the account and see that there's enough, but before making the actual withdrawal, the other partner checks the account and *also* sees that there's enough. When the account balance gets to 10, if both partners check it before making the withdrawal, both will think it's okay to withdraw, and the account will be overdrawn by 10!

Here's the code:

```
public class AccountDanger implements Runnable {
  private Account acct = new Account();
  public static void main (String [] args) {
    AccountDanger r = new AccountDanger();
    Thread one = new Thread(r);
    Thread two = new Thread(r);
    one.setName("Fred");
    two.setName("Lucy");
    one.start();
    two.start();
  }
  public void run() {
    for (int x = 0; x < 5; x++) {
      makeWithdrawal(10);
      if (acct.getBalance() < 0) {
        System.out.println("account is overdrawn!");
      }}}
  private void makeWithdrawal(int amt) {
    if (acct.getBalance() >= amt) {
      System.out.println(Thread.currentThread().getName()
                         + " is going to withdraw");
      try {
        Thread.sleep(500);
      } catch(InterruptedException ex) { }
      acct.withdraw(amt);
      System.out.println(Thread.currentThread().getName()
                         + " completes the withdrawal");
    } else {
      System.out.println("Not enough in account for "
                         + Thread.currentThread().getName()
                         + " to withdraw " + acct.getBalance());
    }}}
```

(Note: You might have to tweak this code a bit on your machine to the "account overdrawn" behavior. You might try much shorter sleep times; you might try adding a `sleep` to the `run()` method... In any case, experimenting will help you lock in the concepts.) So what happened? Is it possible that, say, Lucy checked the balance, fell asleep, Fred checked the balance, Lucy woke up and completed *her* withdrawal, then Fred completes *his* withdrawal, and in the end, they overdraw the account? Look at the (numbered) output:

```
% java AccountDanger
 1. Fred is going to withdraw
 2. Lucy is going to withdraw
 3. Fred completes the withdrawal
 4. Fred is going to withdraw
 5. Lucy completes the withdrawal
 6. Lucy is going to withdraw
 7. Fred completes the withdrawal
 8. Fred is going to withdraw
 9. Lucy completes the withdrawal
10. Lucy is going to withdraw
11. Fred completes the withdrawal
12. Not enough in account for Fred to withdraw 0
13. Not enough in account for Fred to withdraw 0
14. Lucy completes the withdrawal
15. account is overdrawn!
16. Not enough in account for Lucy to withdraw -10
17. account is overdrawn!
18. Not enough in account for Lucy to withdraw -10
19. account is overdrawn!
```

Although each time you run this code the output might be a little different, let's walk

through this particular example using the numbered lines of output. For the first four attempts, everything is fine. Fred checks the balance on line 1 and finds it's okay. At line 2, Lucy checks the balance and finds it okay. At line 3, Fred makes his withdrawal. At this point, the balance Lucy checked for (and believes is still accurate) has actually changed since she last checked. And now Fred checks the balance *again,* before Lucy even completes her first withdrawal. By this point, even Fred is seeing a potentially inaccurate balance because we know Lucy is going to complete her withdrawal. It is possible, of course, that Fred will complete his before Lucy does, but that's not what happens here.

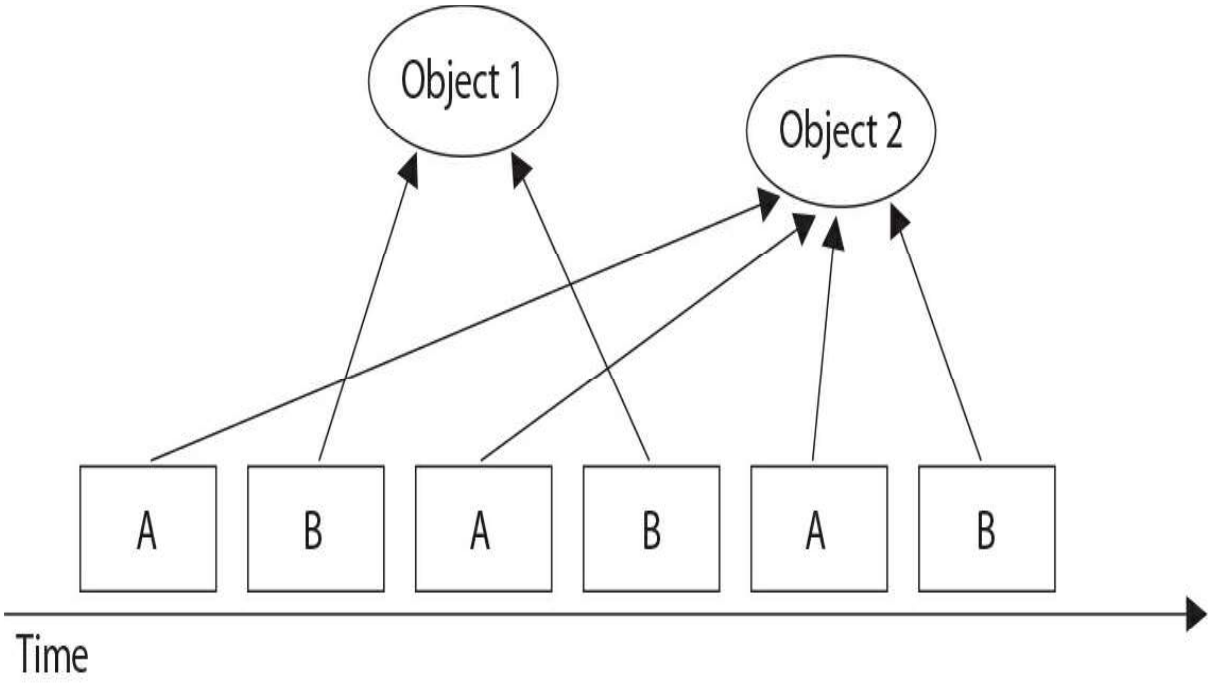
On line 5, Lucy completes her withdrawal and then, before Fred completes his, Lucy does another check on the account on line 6. And so it continues until we get to line 8, where Fred checks the balance and sees that it's 20. On line 9, Lucy completes a withdrawal that she had checked for earlier, and this takes the balance to 10. On line 10, Lucy checks again, sees that the balance is 10, so she knows she can do a withdrawal. *But she didn't know that Fred, too, has already checked the balance on line 8 so he thinks it's safe to do the withdrawal!* On line 11, Fred completes the withdrawal he approved on line 8. This takes the balance to 0. But Lucy still has a pending withdrawal that she got approval for on line 10! You know what's coming.

On lines 12 and 13, Fred checks the balance and finds that there's not enough in the account. But on line 14, Lucy completes her withdrawal and BOOM! The account is now overdrawn by 10—*something we thought we were preventing by doing a balance check prior to a withdrawal.*

Figure 10-4 shows the timeline of what can happen when two threads concurrently access the same object.

FIGURE 10-4

Problems with concurrent access



Thread A will access Object 2 only

Thread B will access Object 1, and then Object 2

This problem is known as a "race condition," where multiple threads can access the same resource (typically an object's instance variables) and can produce corrupted data if one thread "races in" too quickly before an operation that should be "atomic" has completed.

Preventing the Account Overdraw

So what can be done? The solution is actually quite simple. We must guarantee that the two steps of the withdrawal—*checking* the balance and *making* the withdrawal—are never split apart. We need them to always be performed as one operation, even when the thread falls asleep in between step 1 and step 2! We call this an "atomic operation" (although the physics is a little outdated—in this case, "atomic" means "indivisible") because the operation, regardless of the number of actual statements (or underlying bytecode instructions), is completed *before* any other thread code that acts on the same data.

You can't guarantee that a single thread will stay running throughout the entire atomic operation. But you can guarantee that even if the thread running the atomic operation moves in and out of the running state, no other running thread will be able to act on the same data. In other words, if Lucy falls asleep after checking the balance, we can stop Fred from checking the balance until after Lucy wakes up and completes her withdrawal.

So how do you protect the data? You must do two things:

- Mark the variables `private`.
- Synchronize the code that modifies the variables.

Remember, you protect the variables in the normal way—using an access control modifier. It's the method code that you must protect so only one thread at a time can be executing that code. You do this with the `synchronized` keyword.

We can solve all of Fred and Lucy's problems by adding one word to the code. We mark the `makeWithdrawal()` method `synchronized` as follows:

```
private synchronized void makeWithdrawal(int amt) {
  if (acct.getBalance() >= amt) {
    System.out.println(Thread.currentThread().getName() +
                       " is going to withdraw");
    try {
      Thread.sleep(500);
    } catch(InterruptedException ex) { }
    acct.withdraw(amt);
    System.out.println(Thread.currentThread().getName() +
                        " completes the withdrawal");
  } else {
    System.out.println("Not enough in account for "
                      + Thread.currentThread().getName()
                      + " to withdraw " + acct.getBalance());
  }
}
```

Now we've guaranteed that once a thread (Lucy or Fred) starts the withdrawal process by invoking `makeWithdrawal()`, the other thread cannot enter that method until the first one completes the process by exiting the method. The new output shows the benefit of synchronizing the `makeWithdrawal()` method:

```
% java AccountDanger
Fred is going to withdraw
Fred completes the withdrawal
Lucy is going to withdraw
Lucy completes the withdrawal
Fred is going to withdraw
Fred completes the withdrawal
Lucy is going to withdraw
Lucy completes the withdrawal
Fred is going to withdraw
Fred completes the withdrawal
Not enough in account for Lucy to withdraw 0
Not enough in account for Fred to withdraw 0
Not enough in account for Lucy to withdraw 0
Not enough in account for Fred to withdraw 0
Not enough in account for Lucy to withdraw 0
```

Notice that now both threads, Lucy and Fred, always check the account balance *and* complete the withdrawal before the other thread can check the balance.

Synchronization and Locks

How does synchronization work? With locks. Every object in Java has a built-in lock that only comes into play when the object has synchronized method code. When we enter a synchronized non-`static` method, we automatically acquire the lock associated with the current instance of the class whose code we're executing (the `this` instance). Acquiring a lock for an object is also known as getting the lock, or locking the object, locking *on* the object, or synchronizing on the object. We may also use the term *monitor* to refer to the object whose lock we're acquiring. Technically, the lock and the monitor are two different things, but most people talk about the two interchangeably, and we will too.

Since there is only one lock per object, if one thread has picked up the lock, no other thread can pick up the lock until the first thread releases (or returns) the lock. This means no other thread can enter the synchronized code (which means it can't enter any `synchronized` method of that object) until the lock has been released. Typically,

releasing a lock means the thread holding the lock (in other words, the thread currently in the `synchronized` method) exits the `synchronized` method. At that point, the lock is free until some other thread enters a `synchronized` method on that object. Remember the following key points about locking and synchronization:

- Only methods (or blocks) can be `synchronized`, not variables or classes.
- Each object has just one lock.
- Not all methods in a class need to be `synchronized`. A class can have both `synchronized` and non-`synchronized` methods.
- If two threads are about to execute a `synchronized` method in a class and both threads are using the same instance of the class to invoke the method, only one thread at a time will be able to execute the method. The other thread will need to wait until the first one finishes its method call. In other words, once a thread acquires the lock on an object, no other thread can enter any of the `synchronized` methods in that class (for that object).
- If a class has both `synchronized` and non-`synchronized` methods, multiple threads can still access the class's non-`synchronized` methods! If you have methods that don't access the data you're trying to protect, then you don't need to synchronize them. Synchronization can cause a hit in some cases (or even deadlock if used incorrectly), so you should be careful not to overuse it.
- If a thread goes to sleep, it holds any locks it has—it doesn't release them.
- A thread can acquire more than one lock. For example, a thread can enter a `synchronized` method, thus acquiring a lock, and then immediately invoke a `synchronized` method on a different object, thus acquiring that lock as well. As the stack unwinds, locks are released again. Also, if a thread acquires a lock and then attempts to call a `synchronized` method on that same object, no problem. The JVM knows that this thread already has the lock for this object, so the thread is free to call other `synchronized` methods on the same object, using the lock the thread already has.
- You can synchronize a block of code rather than a method.

Because synchronization does hurt concurrency, you don't want to synchronize any more code than is necessary to protect your data. So if the scope of a method is more than needed, you can reduce the scope of the synchronized part to something less than a full method—to just a block. We call this, strangely, a *synchronized block,* and it looks like this:

```
class SyncTest {
  public void doStuff() {
    System.out.println("not synchronized");
    synchronized(this) {
      System.out.println("synchronized");
    }
  }
}
```

When a thread is executing code from within a `synchronized` block, including any method code invoked from that `synchronized` block, the code is said to be executing in a synchronized context. The real question is, synchronized on what? Or, synchronized on which object's lock?

When you synchronize a method, the object used to invoke the method is the object whose lock must be acquired. But when you synchronize a block of code, you specify which object's lock you want to use as the lock, so you could, for 677example, use some third-party object as the lock for this piece of code. That gives you the ability to have more than one lock for code synchronization within a single object.

Or you can synchronize on the current instance (`this`) as in the previous code. Since that's the same instance that `synchronized` methods lock on, it means you could always replace a `synchronized` method with a non-`synchronized` method containing a `synchronized` block. In other words, this:

```
public synchronized void doStuff() {
  System.out.println("synchronized");
}
```

is equivalent to this:

```
public void doStuff() {
  synchronized(this) {
    System.out.println("synchronized");
  }
}
```

These methods both have the exact same effect—in practical terms. The compiled bytecodes may not be exactly the same for the two methods, but they *could* be—and any differences are not really important. The first form is shorter and more familiar to most people, but the second can be more flexible.

Can Static Methods Be Synchronized?

`static` methods can be `synchronized`. There is only one copy of the static data you're trying to protect, so you only need one lock per class to synchronize `static` methods—a lock for the whole class. There is such a lock; every class loaded in Java has a corresponding instance of `java.lang.Class` representing that class. It's that `java.lang.Class` instance whose lock is used to protect any synchronized `static` methods of the class. There's nothing special you have to do to synchronize a `static` method:

```
public static synchronized int getCount() {
  return count;
}
```

Again, this could be replaced with code that uses a `synchronized` block. If the method is defined in a class called `MyClass`, the equivalent code is as follows:

```
public static int getCount() {
  synchronized(MyClass.class) {
    return count;
  }
}
```

Wait—what's that `MyClass.class` thing? That's called a *class literal.* It's a special feature in the Java language that tells the compiler (who tells the JVM): Go and find me the instance of `Class` that represents the class called `MyClass`. You can also do this with the following code:

```
public static void classMethod() throws ClassNotFoundException {
  Class cl = Class.forName("MyClass");
  synchronized (cl) {
    // do stuff
  }
}
```

However, that's longer, ickier, and most importantly, *not on the OCP exam*. But it's quick and easy to use a class literal—just write the name of the class and add `.class` at the end. No quotation marks needed. Now you've got an expression for the `Class` object you need to synchronize on.

EXERCISE 10-2

Synchronizing a Block of Code

In this exercise, we will attempt to synchronize a block of code. Within that block of code, we will get the lock on an object so that other threads cannot modify it while the block of code is executing. We will be creating three threads that will all attempt to manipulate the same object. Each thread will output a single letter 100 times and then increment that letter by one. The object we will be using is `StringBuffer`.

We could synchronize on a `String` object, but strings cannot be modified once they are created, so we would not be able to increment the letter without generating a new `String` object. The final output should have 100 *A*s, 100 *B*s, and 100 *C*s, all in unbroken lines.

1. Create a class and extend the `Thread` class.
2. Override the `run()` method of `Thread`. This is where the `synchronized` block of code will go.
3. For our three thread objects to share the same object, we will need to create a constructor that accepts a `StringBuffer` object in the argument.
4. The `synchronized` block of code will obtain a lock on the `StringBuffer` object from step 3.
5. Within the block, output the `StringBuffer` 100 times and then increment the letter in the `StringBuffer`.
6. Finally, in the `main()` method, create a single `StringBuffer` object using the letter *A,* then create three instances of our class and start all three of them.

What Happens If a Thread Can't Get the Lock?

If a thread tries to enter a `synchronized` method and the lock is already taken, the thread is said to be blocked on the object's lock. Essentially, the thread goes into a kind of pool for that particular object and has to sit there until the lock is released and the thread can again become runnable/running. Just because a lock is released doesn't mean any particular thread will get it. There might be three threads waiting for a single lock, for example, and there's no guarantee that the thread that has waited the longest will get the lock first.

When thinking about blocking, it's important to pay attention to which objects are being used for locking:

- Threads calling non-`static synchronized` methods in the same class will only block each other if they're invoked using the same instance. That's because they each lock on `this` instance, and if they're called using two different instances, they get two locks, which do not interfere with each other.
- Threads calling `static synchronized` methods in the same class will always block each other—they all lock on the same `Class` instance.
- A `static synchronized` method and a non-`static synchronized` method will not block each other, ever. The `static` method locks on a `Class` instance, while the non-`static` method locks on the `this` instance—these actions do not interfere with each other at all.
- For `synchronized` blocks, you have to look at exactly what object has been used for locking. (What's inside the parentheses after the word `synchronized`?) Threads that synchronize on the same object will block each other. Threads that synchronize on different objects will not.

Table 10-1 lists the thread-related methods and whether the thread gives up its lock as a result of the call.

So When Do I Need to Synchronize?

TABLE 10-1

Methods and Lock Status

Give Up Locks	Keep Locks	Class Defining the Method
`wait()`	`notify()` (Although the thread will probably exit the synchronized code shortly after this call and thus give up its locks)	`java.lang.Object`
	`join()`	`java.lang.Thread`
	`sleep()`	`java.lang.Thread`
	`yield()`	`java.lang.Thread`

Synchronization can get pretty complicated, and you may be wondering why you would want to do this at all if you can help it. But remember the earlier "race conditions" example with Lucy and Fred making withdrawals from their account. When we use threads, we usually need to use some synchronization somewhere to make sure our methods don't interrupt each other at the wrong time and mess up our data. Generally, any time more than one thread is accessing mutable (changeable) data, you synchronize to protect that data to make sure two threads aren't changing it at the same time (or that one isn't changing it at the same time the other is reading it, which is also confusing). You don't need to worry about local variables—each thread gets its own copy of a local variable. Two threads executing the same method at the same time will use different copies of the local variables, and they won't bother each other. However, you do need to worry about `static` and non-`static` fields if they contain data that can be changed.

For changeable data in a non-`static` field, you usually use a non-`static` method to access it. By synchronizing that method, you will ensure that any threads trying to run that method *using the same instance* will be prevented from simultaneous access. But a thread working with a *different* instance will not be affected because it's acquiring a lock on the other instance. That's what we want—threads working with the same data need to go one at a time, but threads working with different data can just ignore each other and run whenever they want to; it doesn't matter.

For changeable data in a `static` field, you usually use a `static` method to access it. And again, by synchronizing the method, you ensure that any two threads trying to access the data will be prevented from simultaneous access, because both threads will have to acquire locks on the `Class` object for the class the `static` method's defined in. Again,

that's what we want.

However—what if you have a non-`static` method that accesses a `static` field? Or a `static` method that accesses a non-`static` field (using an instance)? In these cases, things start to get messy quickly, and there's a very good chance that things will not work the way you want. If you've got a `static` method accessing a non-`static` field and you synchronize the method, you acquire a lock on the Class object. But what if there's another method that also accesses the non-`static` field, this time using a non-`static` method? It probably synchronizes on the current instance (`this`) instead. Remember that a `static synchronized` method and a non-`static synchronized` method will not block each other—they can run at the same time. Similarly, if you access a `static` field using a non-`static` method, two threads might invoke that method using two different `this` instances. Which means they won't block each other because they use different locks. Which means two threads are simultaneously accessing the same `static` field—exactly the sort of thing we're trying to prevent.

It gets very confusing trying to imagine all the weird things that can happen here. To keep things simple, in order to make a class thread-safe, methods that access changeable fields need to be `synchronized`.

Access to `static` fields should be done using `static synchronized` methods. Access to non-`static` fields should be done using non-`static synchronized` methods, for example:

```
public class Thing {
  private static int staticField;
  private int nonstaticField;
  public static synchronized int getStaticField() {
    return staticField;
  }
  public static synchronized void setStaticField(
                                          int staticField) {
    Thing.staticField = staticField;
  }
  public synchronized int getNonstaticField() {
    return nonstaticField;
  }
  public synchronized void setNonstaticField(
                                          int nonstaticField) {
    this.nonstaticField = nonstaticField;
  }
}
```

What if you need to access both `static` and `non-static` fields in a method? Well, there are ways to do that, but it's beyond what you need for the exam. You will live a longer, happier life if you JUST DON'T DO IT. Really. Would we lie?

Thread-Safe Classes

When a class has been carefully synchronized to protect its data (using the rules just given or using more complicated alternatives), we say the class is "thread-safe." Many classes in the Java APIs already use synchronization internally in order to make the class "thread-safe." For example, `StringBuffer` and `StringBuilder` are nearly identical classes, except that all the methods in `StringBuffer` are `synchronized` when necessary, whereas those in `StringBuilder` are not. Generally, this makes `StringBuffer` safe to use in a multithreaded environment, whereas `StringBuilder` is not. (In return, `StringBuilder` is a little bit faster because it doesn't bother synchronizing.) However,

even when a class is "thread-safe," it is often dangerous to rely on these classes to provide the thread protection you need. (C'mon, the repeated quotes used around "thread-safe" had to be a clue, right?) You still need to think carefully about how you use these classes. As an example, consider the following class:

```
import java.util.*;
public class NameList {
  private List<String> names = Collections.synchronizedList(
                                          new LinkedList<>());
  public void add(String name) {
    names.add(name);
  }
  public String removeFirst() {
    if (names.size() > 0)
      return (String) names.remove(0);
    else
      return null;
  }
}
```

The method `Collections.synchronizedList()` returns a `List` whose methods are all `synchronized` and "thread-safe" according to the documentation (like a Vector—but since this is the 21st century, we're not going to use a Vector here). The question is, can the `NameList` class be used safely from multiple threads? It's tempting to think that yes, because the data in names is in a synchronized collection, the `NameList` class is "safe" too. However, that's not the case—the `removeFirst()` may sometimes throw an `IndexOutOfBoundsException`. What's the problem? Doesn't it correctly check the `size()` of `names` before removing anything to make sure there's something there? How could this code fail? Let's try to use `NameList` like this:

```
public static void main(String[] args) {
  final NameList nl = new NameList();
  nl.add("Ozymandias");
  class NameDropper extends Thread {
    public void run() {
      String name = nl.removeFirst();
      System.out.println(name);
    }
  }
  Thread t1 = new NameDropper();
  Thread t2 = new NameDropper();
  t1.start();
  t2.start();
}
```

What might happen here is that one of the threads will remove the one name and print it, and then the other will try to remove a name and get `null`. If we think just about the calls to `names.size()` and `names.remove(0)`, they occur in this order:

Thread `t1` executes `names.size()`, which returns `1`.
Thread `t1` executes `names.remove(0)`, which returns `Ozymandias`.
Thread `t2` executes `names.size()`, which returns `0`.
Thread `t2` does not call `remove(0)`.

The output here is

```
Ozymandias
null
```

However, if we run the program again, something different might happen:

Thread `t1` executes `names.size()`, which returns `1`.
Thread `t2` executes `names.size()`, which returns `1`.
Thread `t1` executes `names.remove(0)`, which returns `Ozymandias`.
Thread `t2` executes `names.remove(0)`, which throws an exception because the list is now empty.

The thing to realize here is that in a "thread-safe" class like the one returned by `synchronizedList()`, each *individual* method is synchronized. So `names.size()` is `synchronized`, and `names.remove(0)` is `synchronized`. But nothing prevents another thread from doing something else to the list *in between* those two calls. And that's where problems can happen.

There's a solution here: Don't rely on `Collections.synchronizedList()`. Instead, synchronize the code yourself:

```
import java.util.*;
public class NameList {
  private List names = new LinkedList();
  public synchronized void add(String name) {
    names.add(name);
  }
  public synchronized String removeFirst() {
    if (names.size() > 0)
      return (String) names.remove(0);
    else
      return null;
  }
}
```

Now the entire `removeFirst()` method is `synchronized`, and once one thread starts it and calls `names.size()`, there's no way the other thread can cut in and steal the last name. The other thread will just have to wait until the first thread completes the `removeFirst()` method.

The moral here is that just because a class is described as "thread-safe" doesn't mean it is *always* thread-safe. If individual methods are synchronized, that may not be enough—you may be better off putting in synchronization at a higher level (i.e., put it in the block or method that *calls* the other methods). Once you do that, the original synchronization (in this case, the synchronization inside the object returned by `Collections.synchronizedList()`) may well become redundant.

Thread Deadlock

Perhaps the scariest thing that can happen to a Java program is deadlock. Deadlock occurs when two threads are blocked, with each waiting for the other's lock. Neither can run until the other gives up its lock, so they'll sit there forever.

This can happen, for example, when thread A hits `synchronized` code, acquires a lock B, and then enters another method (still within the `synchronized` code it has the lock on) that's also `synchronized`. But thread A can't get the lock to enter this `synchronized` code—block C—because another thread D has the lock already. So thread A goes off to the waiting-for-the-C-lock pool, hoping that thread D will hurry up and release the lock (by completing the `synchronized` method). But thread A will wait a very long time indeed, because while thread D picked up lock C, it then entered a method `synchronized` on lock B. Obviously, thread D can't get the lock B because thread A has it. And thread A won't release it until thread D releases lock C. But thread D won't release lock C until after it can get lock B and continue. And there they sit. The following example demonstrates deadlock:

```
 1. public class DeadlockRisk {
 2.  private static class Resource {
 3.    public int value;
 4.  }
 5.  private Resource resourceA = new Resource();
 6.  private Resource resourceB = new Resource();
 7.  public int read() {
 8.    synchronized(resourceA) { // May deadlock here
 9.      synchronized(resourceB) {
10.        return resourceB.value + resourceA.value;
11.      }
12.    }
13.  }
14.
15.  public void write(int a, int b) {
16.    synchronized(resourceB) { // May deadlock here
```

```
17.        synchronized(resourceA) {
18.          resourceA.value = a;
19.          resourceB.value = b;
20.        }
21.      }
22.    }
23.  }
```

Assume that `read()` is started by one thread and `write()` is started by another. If there are two different threads that may read and write independently, there is a risk of deadlock at line 8 or 16. The reader thread will have `resourceA`, the writer thread will have `resourceB`, and both will get stuck waiting for the other.

Code like this almost never results in deadlock because the CPU has to switch from the reader thread to the writer thread at a particular point in the code, and the chances of deadlock occurring are quite small. The application may work fine 99.9 percent of the time.

The preceding simple example is easy to fix; just swap the order of locking for either the reader or the writer at lines 16 and 17 (or lines 8 and 9). More complex deadlock situations can take a long time to figure out.

Regardless of how little chance there is for your code to deadlock, the bottom line is that if you deadlock, you're dead. There are design approaches that can help avoid deadlock, including strategies for always acquiring locks in a predetermined order.

But that's for you to study and is beyond the scope of this book. We're just trying to get you through the exam. If you learn everything in this chapter, though, you'll still know more about threads than most experienced Java programmers.

Thread Livelock

Livelock is almost as scary to a Java program as deadlock. Livelock is similar to deadlock, except that the threads aren't officially dead; they're just too busy to make progress.

Imagine you've got a program with two threads and two locks. Each thread must get both locks in order to proceed with the work it needs to do. Thread one successfully acquires lock 1 and then tries to get lock 1 and fails, because thread 2 has already gotten lock 2. Thread 1 then unlocks lock 1, giving thread 2 a chance to get it, waits a little while and then tries to get lock 1 and lock 2 again.

At the same time, thread 2 gets lock 2 and then attempts to get lock 1. Well, thread 1 already has lock 1, so lock 2 does a similar thing: it unlocks lock 2, giving thread 1 a chance to get it, waits a little while, and then tries to get lock 2 and lock 1 again.

Livelock occurs if thread 1 tries to get lock 2 when thread 2 has lock 2, and thread 2 tries to get lock 1 when thread 1 has lock 1; they both free up the locks they have, so then thread 1 and thread 2 are both waiting on the other thread to get the lock each wants and then both get their locks again and go back to trying to get the lock the other thread has… over and over and over again.

In this situation, the threads are not completely deadlocked, but they are making no progress. This is an extremely tricky problem to detect! The timing has to be just right, so you might find your code runs perfectly fine 99 percent of the time and livelocks 1 percent of the time. Fortunately, Java makes livelock hard to do; with ReentrantLock (more on this in Chapter 11 where we talk about concurrency) and careful ordering of how your threads attempt to access locks, you will be unlikely to encounter this problem.

Thread Starvation

Starvation is related to livelock. Starvation is when a thread is unable to make progress because it cannot get access to a shared resource that other threads are hogging. This could happen when another thread gets access to a synchronized resource and then goes into an infinite loop or takes a really long time to use the resource. It could also happen if one thread has higher priority than another thread so the first thread always gets a resource when both threads attempt to access that resource at the same time.

If you are not fiddling with thread priorities—and you are careful about how long a thread can keep access to a resource before yielding or timing out—then you should be able to avoid starvation. Good thread schedulers will help prevent starvation by allocating time fairly between threads behind the scenes.

Race Conditions

A race condition is another scenario that can crop up when working with multiple threads. To understand what a race condition is and how it can occur, let's revisit the singleton pattern from Chapter 2.

In that chapter, we mentioned that our singleton implementation is not thread-safe without some precautions. Let's take a look at how we might use the `Show` singleton from that chapter in a multithreaded program to see where things can go wrong. In the process, we'll create a race condition.

```
public class Show {
    private static Show INSTANCE;
    private Set<String> availableSeats;
```

```
    public static Show getInstance() {   // create a singleton instance
        if (INSTANCE == null) {
              INSTANCE = new Show();      // should be only one Show!
        }
        return INSTANCE;
    }
    private Show() {
        availableSeats = new HashSet<String>();
        availableSeats.add("1A");
        availableSeats.add("1B");
    }
    public boolean bookSeat(String seat) {
        return availableSeats.remove(seat);
    }
}

public class TestShow {
    public static void main(String[] args) {
        TestShow testThreads = new TestShow();
        testThreads.go();
    }
    public void go() {
        // create Thread 1, which will try to book seats 1A and 1B
        Thread getSeats1 = new Thread(() -> {
            ticketAgentBooks("1A");
            ticketAgentBooks("1B");
        });
        // create Thread 2, which will try to book seats 1A and 1B
        Thread getSeats2 = new Thread(() -> {
            ticketAgentBooks("1A");
            ticketAgentBooks("1B");
        });
        // start both threads
        getSeats1.start();
        getSeats2.start();
    }
    public void ticketAgentBooks(String seat) {
        // get the one instance of the Show Singleton
        Show show = Show.getInstance();
        // book a seat and print
        System.out.println(Thread.currentThread().getName() + ": "
            + show.bookSeat(seat));
    }
}
```

When we run the code, here's what we get:

```
Thread-1: true
Thread-1: true
Thread-0: true
Thread-0: false
```

Uh oh. It looks like we've sold seat 1A twice! That means two people will show up for the concert and expect to get the same seat.

This issue is caused by a *race condition*. A race condition is when two or more threads try to access and change a shared resource at the same time, and the result is dependent on the order in which the code is executed by the threads.

Let's step through the code and see how this happens. In `TestShow`, in `go()` (which we call from `main()`), we create two threads. Each thread tries to book seats 1A and 1B by calling `ticketAgentBooks()`. We start both threads.

Imagine a scenario where thread one calls `ticketAgentBooks()` with the argument "1A". Thread one calls `Show.getInstance()`. It executes the code:

```
if (INSTANCE == null)
```

and determines the value in `INSTANCE` is, indeed, `null`, and so is just about to execute the next line of code when BOOM! The thread is descheduled, and thread two begins executing.

Now, thread two calls `ticketAgentBooks()` with the argument "1A". Thread two calls `Show.getInstance()`. It executes the code:

```
if (INSTANCE == null)
```

and determines the value of INSTANCE is, indeed, null, and so executes the next line of code:

```
INSTANCE = new Show();
```

Thread two then gets descheduled and thread one begins executing again. It then executes the line:

```
INSTANCE = new Show();
```

Now we have two instances of `Show`. So when thread one then uses its instance of `Show` to book a seat by calling `show.bookSeat()` on "1A", it succeeds. Similarly, when thread two uses its instance of `Show` to book a seat, it too succeeds. We had two threads racing to

get what should have been one shared resource, and because of timing, we end up with two instances of a resource and a failure in our program logic.

We can fix this race condition by making the `getInstance()` method `synchronized` and the `INSTANCE` variable `volatile`:

```
private static volatile Show INSTANCE;
public static synchronized Show getInstance() {
  if (INSTANCE == null) {
    INSTANCE = new Show();
  }
  return INSTANCE;
}
```

The `volatile` keyword makes sure that the variable `INSTANCE` is atomic; that is, a write to the variable happens all at once. Nothing can interrupt this process: it either happens completely, or it doesn't happen at all. So in `getInstance()`, where we create a new instance of `Show()` and assign it to `INSTANCE`, that entire operation must complete before the thread can be interrupted.

The *synchronized* keyword makes sure only one thread at a time can access the `getInstance()` method. That ensures we won't get a race condition: in other words, we can't check the value of `INSTANCE` in one thread, then stop, and do the same in another thread.

You can run this code, but you might find it still doesn't work! Why?

Take a look at the method `bookSeat()` in `Show`. When we book a seat by calling this method from the `ticketAgentBooks()` method, we are changing another shared resource, the `availableSeats Set`. However, the `Set` is not thread-safe! That means a thread could be interrupted in the middle of removing seat "1A" from the `Set`, and another thread could come along and access the `Set`. If the operations to remove the seat from the `availableSeats Set` get interleaved, we can still end up in a situation where two threads can book the same seat.

To solve this problem, we must synchronize the `bookSeat()` method, too:

```
public synchronized boolean bookSeat(String seat) {
   return availableSeats.remove(seat);
}
```

Now when we run the code, we should find that each seat can be booked by only one

thread:

```
Thread-0: true
Thread-1: false
Thread-0: true
Thread-1: false
```

Notice that this does not mean that a thread will get both seats. Right? A thread can still be interrupted in between the two seat bookings:

```
Thread-1: false
Thread-1: true
Thread-0: true
Thread-0: false
```

If you want to make sure you sell both seats to the same thread, you have to do even more work. But that's enough for now (and enough about race conditions for the exam).

In a multithreaded environment, we need to make sure our code is designed so it's not dependent on the ordering of the threads. In situations where our code is dependent on ordering or dependent on certain operations not being interrupted, we can get race conditions. As you've seen, there are ways to fix race conditions, but like deadlock, livelock, and starvation, they can be tricky to detect.

You've seen an example where a race condition led to two threads both getting seat "1A". There are also race conditions in which neither thread gets seat "1A". It's far better for neither thread to end up with a seat than for two threads to end up with the same seat (while the venue operator might be unhappy having an empty, unsold seat, it's worse to have customers fighting over seats!). However, by fixing the race conditions in this code, we've enabled the seats to be sold to the threads properly, so only one thread gets any given seat.

CERTIFICATION OBJECTIVE

Thread Interaction (OCP Objectives 10.2 and 10.3)

10.2 Identify potential threading problems among deadlock, starvation, livelock, and race conditions.

10.3 Use synchronized keyword and java.util.concurrent.atomic package to control the order of thread execution.

The last thing we need to look at is how threads can interact with one another to communicate about—among other things—their locking status. The `Object` class has three methods, `wait()`, `notify()`, and `notifyAll()`, that help threads communicate the status of an event that the threads care about. For example, if one thread is a mail-delivery thread and one thread is a mail-processor thread, the mail-processor thread has to keep checking to see if there's any mail to process. Using the wait and notify mechanism, the mail-processor thread could check for mail, and if it doesn't find any, it can say, "Hey, I'm not going to waste my time checking for mail every two seconds. I'm going to go hang out, and when the mail deliverer puts something in the mailbox, have him notify me so I can go back to runnable and do some work." In other words, using `wait()` and `notify()` lets one thread put itself into a "waiting room" until some *other* thread notifies it that there's a reason to come back out.

One key point to remember (and keep in mind for the exam) about `wait()`/`notify()` is this:

> *wait(), notify(), and notifyAll() must be called from within a synchronized context! A thread can't invoke a wait() or notify() method on an object unless it owns that object's lock.*

Here we'll present an example of two threads that depend on each other to proceed with their execution, and we'll show how to use `wait()` and `notify()` to make them interact safely at the proper moment.

Think of a computer-controlled machine that cuts pieces of fabric into different shapes and an application that allows users to specify the shape to cut. The current version of the application has one thread, which loops, first asking the user for instructions, and then directs the hardware to cut the requested shape:

```
public void run(){
  while(true){
    // Get shape from user
    // Calculate machine steps from shape
    // Send steps to hardware
  }
}
```

This design is not optimal because the user can't do anything while the machine is busy and while there are other shapes to define. We need to improve the situation.

A simple solution is to separate the processes into two different threads, one of them interacting with the user and another managing the hardware. The user thread sends the instructions to the hardware thread and then goes back to interacting with the user

immediately. The hardware thread receives the instructions from the user thread and starts directing the machine immediately. Both threads use a common object to communicate, which holds the current design being processed.

The following pseudocode shows this design:

```
public void userLoop(){
  while(true){
    // Get shape from user
    // Calculate machine steps from shape
    // Modify common object with new machine steps
  }
}

public void hardwareLoop(){
  while(true){
    // Get steps from common object
    // Send steps to hardware
  }
}
```

The problem now is to get the hardware thread to process the machine steps as soon as they are available. Also, the user thread should not modify them until they have all been sent to the hardware. The solution is to use `wait()` and `notify()` and also to synchronize some of the code.

The methods `wait()` and `notify()`, remember, are instance methods of `Object`. In the same way that every object has a lock, every object can have a list of threads that are waiting for a signal (a notification) from the object. A thread gets on this waiting list by executing the `wait()` method of the target object. From that moment, it doesn't execute any further instructions until the `notify()` method of the target object is called. If many threads are waiting on the same object, only one will be chosen (in no guaranteed order) to proceed with its execution. If there are no threads waiting, then no particular action is taken. Let's take a look at some real code that shows one object waiting for another object to notify it (take note, it is somewhat complex):

```
1. class ThreadA {
2.   public static void main(String [] args) {
3.     ThreadB b = new ThreadB();
4.     b.start();
5.
6.     synchronized(b) {
7.       try {
8.         System.out.println("Waiting for b to complete...");
9.         b.wait();
10.      } catch (InterruptedException e) {}
11.        System.out.println("Total is: " + b.total);
12.      }
13.    }
14. }
15.
16. class ThreadB extends Thread {
17.   int total;
18.
19.   public void run() {
20.     synchronized(this) {
21.       for(int i=0;i<100;i++) {
22.         total += i;
23.       }
24.       notify();
25.     }
26.   }
27. }
```

This program contains two objects with threads: `ThreadA` contains the main thread, and `ThreadB` has a thread that calculates the sum of all numbers from 0 through 99. As soon as line 4 calls the `start()` method, `ThreadA` will continue with the next line of code in its own class, which means it could get to line 11 before `ThreadB` has finished the calculation. To prevent this, we use the `wait()` method in line 9.

Notice in line 6 the code synchronizes itself with the object `b`—this is because in order to call `wait()` on the object, `ThreadA` must own a lock on `b`. For a thread to call `wait()` or `notify()`, the thread has to be the owner of the lock for that object. When the thread waits, it temporarily releases the lock for other threads to use, but it will need it again to continue execution. It's common to find code like this:

```
synchronized(anotherObject) { // this has the lock on anotherObject
  try {
    anotherObject.wait();
      // the thread releases the lock and waits
      // To continue, the thread needs the lock,
      // so it may be blocked until it gets it.
  } catch(InterruptedException e){}
}
```

The preceding code waits until `notify()` is called on `anotherObject`.

```
synchronized(this) { notify(); }
```

This code notifies a single thread currently waiting on the `this` object. The lock can be acquired much earlier in the code, such as in the calling method. Note that if the thread calling `wait()` does not own the lock, it will throw an `IllegalMonitorStateException`. This exception is not a checked exception, so you don't have to *catch* it explicitly. You should always be clear whether a thread has the lock of an object in any given block of code.

Notice in lines 7–10 there is a `try/catch` block around the `wait()` method. A waiting thread can be interrupted in the same way as a sleeping thread, so you have to take care of the exception:

```
try {
  wait();
} catch(InterruptedException e) {
  // Do something about it
}
```

In the next example, the way to use these methods is to have the hardware thread wait on the shape to be available and the user thread to notify after it has written the steps. The machine steps may comprise global steps, such as moving the required fabric to the cutting area, and a number of substeps, such as the direction and length of a cut. As an example, they could be

```
int fabricRoll;
int cuttingSpeed;
Point startingPoint;
float[] directions;
float[] lengths;
etc..
```

It is important that the user thread does not modify the machine steps while the hardware thread is using them, so this reading and writing should be synchronized.

The resulting code would look like this:

```
class Operator extends Thread {
  public void run(){
    while(true){
        // Get shape from user
      synchronized(this){
        // Calculate new machine steps from shape
      notify();
      }
    }
  }
}
class Machine extends Thread {
  Operator operator; // assume this gets initialized
  public void run(){
    while(true){
      synchronized(operator){
        try {
          operator.wait();
          } catch(InterruptedException ie) {}
          // Send machine steps to hardware
        }
    }
  }
}
```

The machine thread, once started, will immediately go into the waiting state and will wait patiently until the operator sends the first notification. At that point, it is the operator thread that owns the lock for the object, so the hardware thread gets stuck for a while. It's

only after the operator thread abandons the `synchronized` block that the hardware thread can really start processing the machine steps.

While one shape is being processed by the hardware, the user may interact with the system and specify another shape to be cut. When the user is finished with the shape and it is time to cut it, the operator thread attempts to enter the `synchronized` block, maybe blocking until the machine thread has finished with the previous machine steps. When the machine thread has finished, it repeats the loop, going again to the waiting state (and therefore releasing the lock). Only then can the operator thread enter the `synchronized` block and overwrite the machine steps with the new ones.

Having two threads is definitely an improvement over having one, although in this implementation, there is still a possibility of making the user wait. A further improvement would be to have many shapes in a queue, thereby reducing the possibility of requiring the user to wait for the hardware.

There is also a second form of `wait()` that accepts a number of milliseconds as a maximum time to wait. If the thread is not interrupted, it will continue normally whenever it is notified or the specified timeout has elapsed. This normal continuation consists of getting out of the waiting state, but to continue execution, it will have to get the lock for the object:

```
synchronized(a){  // The thread gets the lock on 'a'
  a.wait(2000);   // Thread releases the lock and waits for notify
                  // only for a maximum of two seconds, then goes back
                  // to Runnable
                  // The thread reacquires the lock
                  // More instructions here
}
```

When the **`wait()`** *method is invoked on an object, the thread executing that code gives up its lock on the object immediately. However, when* **`notify()`** *is called, that doesn't mean the thread gives up its lock at that moment. If the thread is still completing synchronized code, the lock is not released until the thread moves out of synchronized code. So just because* **`notify()`** *is called, this doesn't mean the lock becomes available at that moment.*

Using notifyAll( ) When Many Threads May Be Waiting

In most scenarios, it's preferable to notify *all* of the threads that are waiting on a particular object. If so, you can use `notifyAll()` on the object to let all the threads rush out of the waiting area and back to runnable. This is especially important if you have several threads waiting on one object, but for different reasons, and you want to be sure that the *right* thread (along with all of the others) is notified.

```
notifyAll(); // Will notify all waiting threads
```

All of the threads will be notified and start competing to get the lock. As the lock is used and released by each thread, all of them will get into action without a need for further notification.

As we said earlier, an object can have many threads waiting on it, and using `notify()` will affect only one of them. Which one, exactly, is not specified and depends on the JVM implementation, so you should never rely on a particular thread being notified in preference to another.

In cases in which there might be a lot more waiting, the best way to do this is by using `notifyAll()`. Let's take a look at this in some code. In this example, there is one class that performs a calculation and many readers that are waiting to receive the completed calculation. At any given moment, many readers may be waiting.

```
 1. class Reader extends Thread {
 2.   Calculator c;
 3.
 4.   public Reader(Calculator calc) {
 5.     c = calc;
 6.   }
 7.
 8.   public void run() {
 9.     synchronized(c) {
10.       try {
11.        System.out.println("Waiting for calculation...");
12.           c.wait();
13.       } catch (InterruptedException e) {}
14.         System.out.println("Total is: " + c.total);
15.      }
16.    }
17.
18.   public static void main(String [] args) {
19.     Calculator calculator = new Calculator();
20.     new Reader(calculator).start();
21.     new Reader(calculator).start();
22.     new Reader(calculator).start();
23.     new Thread(calculator).start();
24.   }
25. }
26.
27. class Calculator implements Runnable {
28.   int total;
29.
30.   public void run() {
31.     synchronized(this) {
32.       for(int i = 0; i < 100; i++) {
33.         total += i;
34.       }
35.       notifyAll();
36.     }
37.   }
38. }
```

The program starts three threads that are all waiting to receive the finished calculation (lines 18–24) and then starts the calculator with its calculation. Note that if the `run()` method at line 30 used `notify()` instead of `notifyAll()`, only one reader would be notified instead of all the readers.

Using wait( ) in a Loop

Actually, both of the previous examples (Machine/Operator and Reader/Calculator) had a common problem. In each one, there was at least one thread calling `wait()` and another thread calling `notify()` or `notifyAll()`. This works well enough as long as the waiting threads have actually started waiting before the other thread executes the `notify()` or `notifyAll()`. But what happens if, for example, the `Calculator` runs first and calls `notify()` before the `Reader`s have started waiting? This could happen since we can't guarantee the order in which the different parts of the thread will execute. Unfortunately, when the Readers run, they just start waiting right away. They don't do anything to see if the event they're waiting for has already happened. So if the `Calculator` has already called `notifyAll()`, it's not going to call `notifyAll()` again—and the waiting `Reader`s will keep waiting forever. This is probably *not* what the programmer wanted to happen. Almost always, when you want to wait for something, you also need to be able to check if it has already happened. Generally, the best way to solve this is to put in some sort of loop that checks on some sort of conditional expressions and only waits if the thing you're waiting for has not yet happened. Here's a modified, safer version of the earlier fabric-cutting machine example:

```
class Operator extends Thread {
  Machine machine; // assume this gets initialized
  public void run() {
    while (true) {
      Shape shape = getShapeFromUser();
      MachineInstructions job =
                          calculateNewInstructionsFor(shape);
      machine.addJob(job);
    }
  }
}
```

The operaator will still keep on looping forever, getting more shapes from users, calculating new instructions for those shapes, and sending them to the machine. But now the logic for `notify()` has been moved into the `addJob()` method in the `Machine` class:

```
class Machine extends Thread {
  List<MachineInstructions> jobs =
                        new ArrayList<MachineInstructions>();

  public void addJob(MachineInstructions job) {
    synchronized (jobs) {
      jobs.add(job);
      jobs.notify();
```

```
    }
  }
  public void run() {
    while (true) {
      synchronized (jobs) {
        // wait until at least one job is available
        while (jobs.isEmpty()) {
          try {
            jobs.wait();
          } catch (InterruptedException ie) { }
        }
        // If we get here, we know that jobs is not empty
        MachineInstructions instructions = jobs.remove(0);
        // Send machine steps to hardware
      }
    }
  }
}
```

A machine keeps a list of the jobs it's scheduled to do. Whenever an operator adds a new job to the list, it calls the `addJob()` method and adds the new job to the list. Meanwhile, the `run()` method just keeps looping, looking for any jobs on the list. If there are no jobs, it will start waiting. If it's notified, it will stop waiting and then recheck the loop condition: Is the list still empty? In practice, this double-check is probably not necessary, as the only time a `notify()` is ever sent is when a new job has been added to the list. However, it's a good idea to require the thread to recheck the `isEmpty()` condition whenever it's been woken up because it's possible that a thread has accidentally sent an extra `notify()` that was not intended. There's also a possible situation called *spontaneous wakeup* that may exist in some situations—a thread may wake up even though no code has called `notify()` or `notifyAll()`. (At least, no code you know about has called these methods. Sometimes, the JVM may call `notify()` for reasons of its own, or code in some other class calls it for reasons you just don't know.) What this means is that when your thread wakes up from a

`wait()`, you don't know for sure why it was awakened. By putting the `wait()` method in a `while` loop and rechecking the condition that represents what we were waiting for, we ensure that *whatever* the reason we woke up, we will re-enter the `wait()` if (and only if) the thing we were waiting for has not happened yet. In the `Machine` class, the thing we were waiting for is for the jobs list to not be empty. If it's empty, we wait, and if it's not, we don't.

Note also that both the `run()` method and the `addJob()` method synchronize on the same object—the jobs list. This is for two reasons. One is because we're calling `wait()` and `notify()` on this instance, so we need to synchronize in order to avoid an `IllegalMonitorStateException`. The other reason is that 700the data in the jobs list is changeable data stored in a field that is accessed by two different threads. We need to synchronize in order to access that changeable data safely. Fortunately, the same `synchronized` blocks that allow us to `wait()` and `notify()` also provide the required thread safety for our other access to changeable data. In fact, this is a main reason why synchronization is required to use `wait()` and `notify()` in the first place—you almost always need to share some mutable data between threads at the same time, and that means you need synchronization. Notice that the `synchronized` block in `addJob()` is big enough to also include the call to `jobs.add(job)`—which modifies shared data. And the `synchronized` block in `run()` is large enough to include the whole `while` loop—which includes the call to `jobs.isEmpty()`, which accesses shared data.

The moral here is that when you use `wait()` and `notify()` or `notifyAll()`, you should almost always also have a `while` loop around the `wait()` that checks a condition and forces continued waiting until the condition is met. And you should also make use of the required synchronization for the `wait()` and `notify()` calls to also protect whatever other data you're sharing between threads. If you see code that fails to do this, there's usually something wrong with the code—even if you have a hard time seeing what exactly the problem is.

The methods ***wait()***, ***notify()***, *and* ***notifyAll()*** *are methods of only* ***java.lang.Object****, not of* ***java.lang.Thread*** *or* ***java.lang.Runnable****. Be sure you know which methods are defined in* ***Thread****, which in* ***Object****, and which in* ***Runnable*** *(just* ***run()****, so that's an easy one). Of the key methods in* ***Thread****, be sure you know which are* ***static****—****sleep()*** *and* ***yield()****—and which are* ***not static*** *—****join()*** *and* ***start()****. Table 10-2 lists the key methods you'll need to know for the exam, with the* ***static*** *methods shown in italics.*

TABLE 10-2

Key `Thread` Methods

Class Object	Class Thread	Interface Runnable
wait()	start()	run()
notify()	*yield*()	
notifyAll()	*sleep*()	
	join()	

CERTIFICATION SUMMARY

This chapter covered the required thread knowledge you'll need to apply on the certification exam. Threads can be created by either extending the `Thread` class or implementing the `Runnable` interface. The only method that must be implemented in the `Runnable` interface is the `run()` method, but the thread doesn't become a *thread of execution* until somebody calls the `Thread` object's `start()` method. We also looked at how the `sleep()` method can be used to pause a thread, and we saw that when an object goes to sleep, it holds onto any locks it acquired prior to sleeping.

We looked at five thread states: new, runnable, running, blocked/waiting/sleeping, and dead. You learned that when a thread is dead, it can never be restarted even if it's still a valid object on the heap. We saw that there is only one way a thread can transition to running, and that's from runnable. However, once running, a thread can become dead, go to sleep, wait for another thread to finish, block on an object's lock, wait for a notification, or return to runnable.

You saw how two threads acting on the same data can cause serious problems (remember Lucy and Fred's bank account?). We saw that to let one thread execute a method but prevent other threads from running the same object's method, we use the `synchronized` keyword. And we saw how the `wait()`, `notify()`, and `notifyAll()` methods can be used to coordinate activity between different threads.

TWO-MINUTE DRILL

Here are some of the key points from each certification objective in this chapter. Photocopy it and sleep with it under your pillow for complete absorption.

Defining, Instantiating, and Starting Threads (OCP Objective 10.1)

- Threads can be created by extending `Thread` and overriding the `public void run()` method.
- `Thread` objects can also be created by calling the `Thread` constructor that takes a `Runnable` argument. The `Runnable` object is said to be the *target* of the thread.
- A `Runnable` can be defined as an instance of a class that implements the `Runnable` interface. You can create a Runnable with a lambda expression, because `Runnable` is a functional interface.
- You can call `start()` on a `Thread` object only once. If `start()` is called more than once on a `Thread` object, it will throw a `IllegalThreadStateException`.
- It is legal to create many `Thread` objects using the same `Runnable` object as the target.
- When a `Thread` object is created, it does not become a *thread of execution* until its `start()` method is invoked. When a `Thread` object exists but hasn't been started, it is in the *new* state and is not considered *alive*.

Transitioning Between Thread States (OCP Objective 10.1)

- Once a new thread is started, it will always enter the runnable state.
- The thread scheduler can move a thread back and forth between the runnable state and the running state.
- For a single-processor machine, only one thread can be running at a time, although many threads may be in the runnable state.
- There is no guarantee that the order in which threads were started determines the order in which they'll run.
- There's no guarantee that threads will take turns in any fair way. It's up to the thread scheduler, as determined by the particular virtual machine implementation. If you want a guarantee that your threads will take turns, regardless of the underlying JVM, you can use the `sleep()` method. This prevents one thread from hogging the running process while another thread starves. (In most cases, though, `yield()` works well enough to encourage your threads to play together nicely.)
- A running thread may enter a blocked/waiting state by a `wait()`, `sleep()`, or `join()` call.
- A running thread may enter a blocked/waiting state because it can't acquire the lock for a synchronized block of code.
- When the sleep or wait is over, or an object's lock becomes available, the thread can only reenter the runnable state. It will *go* directly from waiting to runnable (well, for

all practical purposes anyway).

- A dead thread cannot be started again.

Sleep, Yield, and Join (OCP Objective 10.1)

- Sleeping is used to delay execution for a period of time, and no locks are released when a thread goes to sleep.
- A sleeping thread is guaranteed to sleep for at least the time specified in the argument to the `sleep()` method (unless it's interrupted), but there is no guarantee as to when the newly awakened thread will actually return to running.
- The `sleep()` method is a `static` method that sleeps the currently executing thread's state. One thread *cannot* tell another thread to sleep.
- The `setPriority()` method gives `Thread` objects a priority of between 1 (low) and 10 (high). Priorities are not guaranteed, and not all JVMs recognize ten distinct priority levels—some levels may be treated as effectively equal.
- If not explicitly set, a thread's priority will have the same priority as the thread that created it.
- The `yield()` method *may* cause a running thread to back out if there are runnable threads of the same priority. There is no guarantee that this will happen, and there is no guarantee that when the thread backs out there will be a *different* thread selected to run. A thread might yield and then immediately reenter the running state.
- The closest thing to a guarantee is that at any given time, when a thread is running, it will usually not have a lower priority than any thread in the runnable state. If a low-priority thread is running when a high-priority thread enters runnable, the JVM will usually preempt the running low-priority thread and put the high-priority thread in.
- When one thread calls the `join()` method of another thread, the currently running thread will wait until the thread it joins with has completed. Think of the `join()` method as saying, "Hey, thread, I want to join on to the end of you. Let me know when you're done, so I can enter the runnable state."

Concurrent Access Problems and Synchronized Threads (OCP Objectives 10.2 and 10.3)

- `synchronized` methods prevent more than one thread from accessing an object's critical method code simultaneously.
- You can use the `synchronized` keyword as a method modifier or to start a synchronized block of code.
- To synchronize a block of code (in other words, a scope smaller than the whole method), you must specify an argument that is the object whose lock you want to

synchronize on.

- While only one thread can be accessing synchronized code of a particular instance, multiple threads can still access the same object's unsynchronized code.
- When a thread goes to sleep, its locks will be unavailable to other threads.
- `static` methods can be `synchronized` using the lock from the `java.lang.Class` instance representing that class.

Communicating with Objects by Waiting and Notifying (OCP Objective 10.1)

- The `wait()` method lets a thread say, "There's nothing for me to do now, so put me in your waiting pool and notify me when something happens that I care about." Basically, a `wait()` call means "let me wait in your pool" or "add me to your waiting list."
- The `notify()` method is used to send a signal to one and only one of the threads that are waiting in that same object's waiting pool.
- The `notify()` method CANNOT specify which waiting thread to notify.
- The method `notifyAll()` works in the same way as `notify()`, only it sends the signal to *all* of the threads waiting on the object.
- All three methods—`wait()`, `notify()`, and `notifyAll()`—must be called from within a `synchronized` context! A thread invokes `wait()` or `notify()` on a particular object, and the thread must currently hold the lock on that object.

Deadlocked, Livelocked, and Starved Threads and Race Conditions (OCP Objective 10.2)

- Deadlocking is when thread execution grinds to a halt because the code is waiting for locks to be removed from objects.
- Deadlocking can occur when a locked object attempts to access another locked object that is trying to access the first locked object. In other words, both threads are waiting for each other's locks to be released; therefore, the locks will *never* be released!
- Deadlocking is bad. Don't do it.
- Livelocking is when thread execution grinds to a halt because the threads are too busy to make any progress. The threads are still working but can't get anywhere.
- Thread starvation is when a thread can't get access to a resource it needs so it starves. It's still alive, but barely.
- A race condition is when two threads race to get the same shared resource, and the result (often wrong) depends on which thread gets there first.

- Race conditions are bad. Don't allow them to happen. Use the `volatile` keyword to protect variables that should be atomic, and the `synchronized` keyword to make sure only one thread at a time can run code that manages a resource.

SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. If you have a rough time with some of these at first, don't beat yourself up. Some of these questions are long and intricate. Expect long and intricate questions on the real exam, too!

1. The following block of code creates a `Thread` using a `Runnable` target:

```
Runnable target = new MyRunnable();
Thread myThread = new Thread(target);
```

   Which of the following classes can be used to create the target so that the preceding code compiles correctly?

A. `public class MyRunnable extends Runnable{public void run(){}}`

B. `public class MyRunnable extends Object{public void run(){}}`

C. `public class MyRunnable implements Runnable{public void run(){}}`

D. `public class MyRunnable implements Runnable{void run(){}}`

E. `public class MyRunnable implements Runnable{public void start(){}}`

2. Given:

```
3. class MyThread extends Thread {
4.    public static void main(String [] args) {
5.       MyThread t = new MyThread();
6.       Thread x = new Thread(t);
7.       x.start();
8.    }
9.    public void run() {
10.      for(int i=0;i<3;++i) {
11.         System.out.print(i + "..");
12.      }
13.   }
14. }
```

What is the result of this code?

A. Compilation fails

B. `1..2..3..`

C. `0..1..2..3..`

D. `0..1..2..`

E. An exception occurs at runtime

3. Given:

```
3. class Test {
4.    public static void main(String [] args) {
5.       printAll(args);
6.    }
7.    public static void printAll(String[] lines) {
8.       for(int i=0;i<lines.length;i++){
9.          System.out.println(lines[i]);
10.         Thread.currentThread().sleep(1000);
11.      }
12.   }
13. }
```

The `static` method `Thread.currentThread()` returns a reference to the currently executing `Thread` object. What is the result of this code?

A. Each `String` in the array `lines` will output, with a one-second pause between lines

B. Each `String` in the array `lines` will output, with no pause in between because this method is not executed in a `Thread`

C. Each `String` in the array `lines` will output, and there is no guarantee that there will be a pause because `currentThread()` may not retrieve this thread

D. This code will not compile

E. Each `String` in the `lines` array will print, with at least a one-second pause between lines

4. Assume you have a class that holds two `private` variables: `a` and `b`. Which of the following pairs can prevent concurrent access problems in that class? (Choose all that apply.)

A.
```
public int read(){return a+b;}
public void set(int a, int b){this.a=a;this.b=b;}
```

B.
```
public synchronized int read(){return a+b;}
public synchronized void set(int a, int b){this.a=a;this.b=b;}
```

C.
```
public int read(){synchronized(a){return a+b;}}
public void set(int a, int b){
    synchronized(a){this.a=a;this.b=b;}}
```

D.
```
public int read(){synchronized(a){return a+b;}}
public void set(int a, int b){
    synchronized(b){this.a=a;this.b=b;}}
```

E.
```
public synchronized(this) int read(){return a+b;}
public synchronized(this) void set(int a, int b){
    this.a=a;this.b=b;}
```

F.
```
public int read(){synchronized(this){return a+b;}}
public void set(int a, int b){
    synchronized(this){this.a=a;this.b=b;}}
```

5. Given:

```
1. public class WaitTest {
2.    public static void main(String [] args) {
3.       System.out.print("1 ");
4.       synchronized(args){
5.          System.out.print("2 ");
6.          try {
7.             args.wait();
8.          }
9.          catch(InterruptedException e){}
10.      }
11.      System.out.print("3 ");
12.   }
13. }
```

What is the result of trying to compile and run this program?

A. It fails to compile because the `IllegalMonitorStateException` of `wait()` is not dealt with in line 7

B. `1 2 3`

C. `1 3`

D. `1 2`

E. At runtime, it throws an `IllegalMonitorStateException` when trying to wait

F. It will fail to compile because it has to be synchronized on the `this` object

6. Assume the following method is properly synchronized and called from a thread A on an object B:

```
wait(2000);
```

After calling this method, when will thread A become a candidate to get another turn at the CPU?

A. After object B is notified, or after two seconds

B. After the lock on B is released, or after two seconds

C. Two seconds after object B is notified

D. Two seconds after lock B is released

7. Which are true? (Choose all that apply.)

   A. The `notifyAll()` method must be called from a synchronized context

   B. To call `wait()`, an object must own the lock on the thread

   C. The `notify()` method is defined in class `java.lang.Thread`

   D. When a thread is waiting as a result of `wait()`, it releases its lock

   E. The `notify()` method causes a thread to immediately release its lock

   F. The difference between `notify()` and `notifyAll()` is that `notifyAll()` notifies all waiting threads, regardless of the object they're waiting on

8. Given this scenario: This class is intended to allow users to write a series of messages so that each message is identified with a timestamp and the name of the thread that wrote the message:

```
public class Logger {
  private StringBuilder contents = new StringBuilder();
  public void log(String message) {
      contents.append(System.currentTimeMillis());
      contents.append(": ");
      contents.append(Thread.currentThread().getName());
      contents.append(message);
      contents.append("\n");
  }
  public String getContents() { return contents.toString(); }
}
```

   How can we ensure that instances of this class can be safely used by multiple threads?

   A. This class is already thread-safe

   B. Replacing `StringBuilder` with `StringBuffer` will make this class thread-safe

   C. Synchronize the `log()` method only

   D. Synchronize the `getContents()` method only

   E. Synchronize both `log()` and `getContents()`

   F. This class cannot be made thread-safe

9. Given:

```
public static synchronized void main(String[] args) throws InterruptedException {
  Thread t = new Thread();
  t.start();
  System.out.print("X");
  t.wait(10000);
  System.out.print("Y");
}
```

What is the result of this code?

A. It prints `X` and exits

B. It prints `X` and never exits

C. It prints `XY` and exits almost immediately

D. It prints `XY` with a 10-second delay between `X` and `Y`

E. It prints `XY` with a 10,000-second delay between `X` and `Y`

F. The code does not compile

G. An exception is thrown at runtime

10. Given:

```
class MyThread extends Thread {
  MyThread() {
    System.out.print("MyThread ");
  }
  public void run() {
    System.out.print("bar ");
  }
  public void run(String s) {
    System.out.print("baz ");
  }
}
public class TestThreads {
  public static void main (String [] args) {
    Thread t = new MyThread() {
      public void run() {
        System.out.print("foo ");
      }
    };
    t.start();
} }
```

What is the result?

A. `foo`

B. `MyThread foo`

C. `MyThread bar`

D. `foo bar`

E. `foo bar baz`

F. `bar foo`

G. Compilation fails

H. An exception is thrown at runtime

11. Given:

```
public class ThreadDemo {
  synchronized void a() { actBusy(); }
  static synchronized void b() { actBusy(); }
  static void actBusy() {
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {}
  }
  public static void main(String[] args) {
    final ThreadDemo x = new ThreadDemo();
    final ThreadDemo y = new ThreadDemo();
    Runnable runnable = () -> {
```

```
            int option = (int) (Math.random() * 4);
            switch (option) {
              case 0: x.a(); break;
              case 0: x.b(); break;
              case 2: y.a(); break;
              case 3: y.b(); break;
            }
        };
        Thread thread1 = new Thread(runnable);
        Thread thread2 = new Thread(runnable);
        thread1.start();
        thread2.start();
    }
}
```

If the code compiles, which of the following pairs of method invocations could NEVER be executing at the same time? (Choose all that apply.)

A. `x.a()` in `thread1`, and `x.a()` in `thread2`

B. `x.a()` in `thread1`, and `x.b()` in `thread2`

C. `x.a()` in `thread1`, and `y.a()` in `thread2`

D. `x.a()` in `thread1`, and `y.b()` in `thread2`

E. `x.b()` in `thread1`, and `x.a()` in `thread2`

F. `x.b()` in `thread1`, and `x.b()` in `thread2`

G. `x.b()` in `thread1`, and `y.a()` in `thread2`

H. `x.b()` in `thread1`, and `y.b()` in `thread2`

I. Compilation fails due to an error in declaring the `Runnable`

12. Given:

```
public class TwoThreads {
  static Thread laurel, hardy;
  public static void main(String[] args) {
    laurel = new Thread() {
      public void run() {
        System.out.println("A");
        try {
          hardy.sleep(1000);
          } catch (Exception e) {
            System.out.println("B");
          }
            System.out.println("C");
          }
        };
        hardy = new Thread() {
          public void run() {
```

```
                System.out.println("D");
                try {
                  laurel.wait();
                } catch (Exception e) {
                    System.out.println("E");
                }
                System.out.println("F");
            }
        };
        laurel.start();
        hardy.start();
    }
}
```

Which letters will eventually appear somewhere in the output? (Choose all that apply.)

A. `A`

B. `B`

C. `C`

D. `D`

E. `E`

F. `F`

G. The answer cannot be reliably determined

H. The code does not compile

13. Given:

```
3. public class Starter implements Runnable {
4.  void go(long id) {
5.    System.out.println(id);
6.  }
7.  public static void main(String[] args) {
8.    System.out.print(Thread.currentThread().getId() + " ");
9.    // insert code here
10. }
11. public void run() { go(Thread.currentThread().getId()); }
12. }
```

And given the following five fragments:

```
I.   new Starter().run();
II.  new Starter().start();
III. new Thread(new Starter());
IV.  new Thread(new Starter()).run();
V.   new Thread(new Starter()).start();
```

When the five fragments are inserted, one at a time at line 9, which are true? (Choose all that apply.)

A. All five will compile

B. Only one might produce the output `4 4`

C. Only one might produce the output `4 2`

D. Exactly two might produce the output `4 4`

E. Exactly two might produce the output `4 2`

F. Exactly three might produce the output `4 4`

G. Exactly three might produce the output `4 2`

14. Given:

```
3. public class Leader implements Runnable {
4.  public static void main(String[] args) {
5.    Thread t = new Thread(new Leader());
6.    t.start();
7.    System.out.print("m1 ");
8.    t.join();
9.    System.out.print("m2 ");
10.  }
11.  public void run() {
12.    System.out.print("r1 ");
13.    System.out.print("r2 ");
14.  }
15. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The output could be `r1 r2 m1 m2`

C. The output could be `m1 m2 r1 r2`

D. The output could be `m1 r1 r2 m2`

E. The output could be `m1 r1 m2 r2`

F. An exception is thrown at runtime

15. Given:

```
 3. class Dudes {
 4.  static long flag = 0;
 5.  // insert code here
 6.    if(flag == 0) flag = id;
 7.    for(int x = 1; x < 3; x++) {
 8.      if(flag == id) System.out.print("yo ");
 9.      else System.out.print("dude ");
10.    }
11.  }
12. }
13. public class DudesChat implements Runnable {
14.  static Dudes d;
15.  public static void main(String[] args) {
16.    new DudesChat().go();
17.  }
18.  void go() {
19.    d = new Dudes();
20.    new Thread(new DudesChat()).start();
21.    new Thread(new DudesChat()).start();
22.  }
23.  public void run() {
24.    d.chat(Thread.currentThread().getId());
25.  }
26. }
```

And given these two fragments:

```
I. synchronized void chat(long id) {
II. void chat(long id) {
```

When fragment I or fragment II is inserted at line 5, which are true? (Choose all that apply.)

A. An exception is thrown at runtime

B. With fragment I, compilation fails

C. With fragment II, compilation fails

D. With fragment I, the output could be `yo dude dude yo`

E. With fragment I, the output could be `dude dude yo yo`

F. With fragment II, the output could be `yo dude dude yo`

16. Given:

```
3. class Chicks {
4.  synchronized void yack(long id) {
5.    for(int x = 1; x < 3; x++) {
6.      System.out.print(id + " ");
7.      Thread.yield();
8.    }
9.  }
10. }
11. public class ChicksYack implements Runnable {
12.  Chicks c;
13.  public static void main(String[] args) {
14.    new ChicksYack().go();
15.  }
16.  void go() {
17.    c = new Chicks();
18.    new Thread(new ChicksYack()).start();
19.    new Thread(new ChicksYack()).start();
20.  }
```

```
21.  public void run() {
22.    c.yack(Thread.currentThread().getId());
23.  }
24. }
```

Which are true? (Choose all that apply.)

A. Compilation fails

B. The output could be `4 4 2 3`

C. The output could be `4 4 2 2`

D. The output could be `4 4 4 2`

E. The output could be `2 2 4 4`

F. An exception is thrown at runtime

17. Given:

```
3. public class Chess implements Runnable {
4.  public void run() {
5.    move(Thread.currentThread().getId());
6.  }
7.  // insert code here
8.    System.out.print(id + " ");
9.    System.out.print(id + " ");
10. }
11. public static void main(String[] args) {
12.   Chess ch = new Chess();
13.   new Thread(ch).start();
14.   new Thread(new Chess()).start();
15. }
16. }
```

And given these two fragments:

```
I. synchronized void move(long id) {
II. void move(long id) {
```

When either fragment I or fragment II is inserted at line 7, which are true? (Choose all that apply.)

A. Compilation fails

B. With fragment I, an exception is thrown

C. With fragment I, the output could be `4 2 4 2`

D. With fragment I, the output could be `4 4 2 3`

E. With fragment II, the output could be `2 4 2 4`

18. You have two threads, t1 and t2, attemping to access a shared resource, and t2 is always descheduled when it tries to access that resource. What is this kind of problem called?

A. A race condition

B. Deadlock

C. Livelock

D. Starvation

E. Synchronization

F. Multitasking

A SELF TEST ANSWERS

1. ☑ C is correct. The class implements the `Runnable` interface with a legal `run()` method.

   ☒ A is incorrect because interfaces are implemented, not extended. B is incorrect because even though the class has a valid `public void run()` method, it does not implement the `Runnable` interface. D is incorrect because the `run()` method must be public. E is incorrect because the method to implement is `run()`, not `start()`. Note that we could replace the first line of code with:

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

   and dispense with `MyRunnable` completely. (OCP Objective 10.1)

2. ☑ D is correct. The thread `MyThread` will start and loop three times (from 0 to 2).

   ☒ A is incorrect because the `Thread` class implements the `Runnable` interface; therefore, in line 6, `Thread` can take an object of type `Thread` as an argument in the constructor (this is NOT recommended). B and C are incorrect because the variable `i` in the `for` loop starts with a value of 0 and ends with a value of 2. E is incorrect based

on the above. (OCP Objective 10.1)

3. ☑ D is correct. The `sleep()` method must be enclosed in a `try/catch` block, or the method `printAll()` must declare it throws the `InterruptedException`.

   ☒ E is incorrect, but it would be correct if the `InterruptedException` was dealt with (A is too precise). B is incorrect (even if the `InterruptedException` was dealt with) because all Java code, including the `main()` method, runs in threads. C is incorrect. The `sleep()` method is `static`; it always affects the currently executing thread. (OCP Objective 10.1)

4. ☑ B and F are correct. By marking the methods as `synchronized`, the threads will get the lock of the `this` object before proceeding. Only one thread will be setting or reading at any given moment, thereby assuring that `read()` always returns the addition of a valid pair.

   ☒ A is incorrect because it is not synchronized; therefore, there is no guarantee that the values added by the `read()` method belong to the same pair. C and D are incorrect; only objects can be used to synchronize on. E is incorrect because it fails to compile—it is not possible to select other objects (even `this`) to synchronize on when declaring a method as `synchronized`. (OCP Objectives 10.2 and 10.3)

5. ☑ D is correct. 1 and 2 will be printed, but there will be no return from the `wait` call because no other thread will notify the main thread, so 3 will never be printed. It's frozen at line 7.

   ☒ A is incorrect; `IllegalMonitorStateException` is an unchecked exception. B and C are incorrect; 3 will never be printed, since this program will wait forever. E is incorrect because `IllegalMonitorStateException` will never be thrown because the `wait()` is done on `args` within a block of code synchronized on `args`. F is incorrect because any object can be used to synchronize on, and `this` and `static` don't mix. (OCP Objective 10.3)

6. ☑ A is correct. Either of the two events will make the thread a candidate for running again.

   ☒ B is incorrect because a waiting thread will not return to runnable when the lock is released unless a notification occurs. C is incorrect because the thread will become a candidate immediately after notification. D is also incorrect because a thread will not come out of a waiting pool just because a lock has been released. (OCP Objective 10.3)

7. ☑ A is correct because `notifyAll()` (and `wait()` and `notify()`) must be called from within a synchronized context. D is a correct statement.

   ☒ B is incorrect because to call `wait()`, the thread must own the lock on the object that `wait()` is being invoked on, not the other way around. C is incorrect because `notify()` is defined in `java.lang.Object`. E is incorrect because `notify()` will

not cause a thread to release its locks. The thread can only release its locks by exiting the synchronized code. F is incorrect because `notifyAll()` notifies all the threads waiting on a particular locked object, not all threads waiting on *any* object. (OCP Objectives 10.2 and 10.3)

8. ☑ E is correct because synchronizing the `public` methods is sufficient to make this safe, which is why F is incorrect. This class is not thread-safe unless some sort of synchronization protects the changing data.

   ☒ B is incorrect because although a `StringBuffer` is synchronized internally, we call `append()` multiple times, and nothing would prevent two simultaneous `log()` calls from mixing up their messages. C and D are incorrect because if one method remains unsynchronized, it can run while the other is executing, which could result in reading the contents while one of the messages is incomplete, or worse. (You don't want to call `toString()` on the `StringBuffer` as it's resizing its internal character array.) F is incorrect based on the information above.(OCP Objective 10.3)

9. ☑ G is correct. The code does not acquire a lock on `t` before calling `t.wait()`, so it throws an `IllegalMonitorStateException`. The method is `synchronized`, but it's not synchronized on `t` so the exception will be thrown. If the wait were placed inside a `synchronized(t)` block, then D would be correct.

   ☒ A, B, C, D, E, and F are incorrect based on the logic described above. (OCP Objective 10.3)

10. ☑ B is correct. In the first line of `main` we're constructing an instance of an anonymous inner class extending from `MyThread`. So the `MyThread` constructor runs and prints `MyThread`. Next, `main()` invokes `start()` on the new thread instance, which causes the overridden `run()` method (the `run()` method in the anonymous inner class) to be invoked.

    ☒ A, C, D, E, F, G, and H are incorrect based on the logic described above. (OCP Objective 10.1)

11. ☑ A, F, and H are correct. A is correct because when `synchronized` instance methods are called on the same *instance,* they block each other. F and H can't happen because `synchronized static` methods in the same class block each other, regardless of which instance was used to call the methods. (An instance is not required to call `static` methods; only the class.)

    ☒ C, although incorrect, could happen because `synchronized` instance methods called on different instances do not block each other. B, D, E, and G are incorrect but also could all happen because instance methods and `static` methods lock on different objects and do not block each other. I is incorrect because the code compiles. (OCP Objectives 10.2 and 10.3)

12. ☑ A, C, D, E, and F are correct. This may look like `laurel` and `hardy` are battling

to cause the other to `sleep()` or `wait()`—but that's not the case. Since `sleep()` is a `static` method, it affects the current thread, which is `laurel` (even though the method is invoked using a reference to `hardy`). That's misleading, but perfectly legal, and the `Thread laurel` is able to sleep with no exception, printing A and C (after at least a one-second delay). Meanwhile, `hardy` tries to call `laurel.wait()`—but `hardy` has not `synchronized` on `laurel`, so calling `laurel.wait()` immediately causes an `IllegalMonitorStateException`, and so `hardy` prints D, E, and F. Although the *order* of the output is somewhat indeterminate (we have no way of knowing whether A is printed before D, for example), it is guaranteed that A, C, D, E, and F will all be printed in some order, eventually—so G is incorrect.

☒ B, G, and H are incorrect based on the above. (OCP Objective 10.2)

13. ☑ C and D are correct. Fragment I doesn't start a new thread. Fragment II doesn't compile. Fragment III creates a new thread but doesn't start it. Fragment IV creates a new thread and invokes `run()` directly, but it doesn't start the new thread. Fragment V creates and starts a new thread.

    ☒ A, B, E, F, and G are incorrect based on the above. (OCP Objective 10.1)

14. ☑ A is correct. The `join()` must be placed in a `try/catch` block. If it were, answers B and D would be correct. The `join()` causes the main thread to pause and join the end of the other thread, meaning `"m2"` must come last.

    ☒ B, C, D, E, and F are incorrect based on the above. (OCP Objective 10.1)

15. ☑ F is correct. With Fragment I, the `chat` method is `synchronized`, so the two threads can't swap back and forth. With either fragment, the first output must be `yo`.

    ☒ A, B, C, D, and E are incorrect based on the above. (OCP Objective 10.3)

16. ☑ F is correct. When `run()` is invoked, it is with a new instance of `ChicksYack` and `c` has not been assigned to an object. If `c` were static, then because `yack` is `synchronized`, answers C and E would have been correct.

    ☒ A, B, C, D, and E are incorrect based on the above. (OCP Objectives 10.1 and 10.3)

17. ☑ C and E are correct. E should be obvious. C is correct because, even though `move()` is `synchronized`, it's being invoked on two different objects.

    ☒ A, B, and D are incorrect based on the above. (OCP Objective 10.3)

18. ☑ D is correct. Starvation occurs when one or more threads cannot get access to a resource.

    ☒ A, B, C, E and F are incorrect based on the above. (OCP Objective 10.2)

EXERCISE ANSWERS

Exercise 10-1: Creating a Thread and Putting It to Sleep

The final code should look something like this:

```
class TheCount extends Thread {
   public void run() {
      for(int i = 1;i<=100;++i) {
         System.out.print(i + "  ");
         if(i % 10 == 0)  System.out.println("Hahaha");
         try { Thread.sleep(1000); }
         catch(InterruptedException e) {}
      }
   }
   public static void main(String [] args) {
      new TheCount().start();
   }
}
```

Exercise 10-2: Synchronizing a Block of Code

Your code might look something like this when completed:

```
class InSync extends Thread {
  StringBuffer letter;
  public InSync(StringBuffer letter) { this.letter = letter; }
  public void run() {
    synchronized(letter) {      // #1
      for(int i = 1;i<=100;++i) System.out.print(letter);
      System.out.println();
      char temp = letter.charAt(0);
      ++temp;         // Increment the letter in StringBuffer:
      letter.setCharAt(0, temp);
    }     // #2
  }
  public static void main(String [] args) {
    StringBuffer sb = new StringBuffer("A");
    new InSync(sb).start();  new InSync(sb).start();
    new InSync(sb).start();
  }
}
```

Just for fun, try removing lines 1 and 2 and then run the program again. It will be unsynchronized—watch what happens.