JAVA™ 2

SUN™ CERTIFIED PROGRAMMER & DEVELOPER

9

Threads

CERTIFICATION OBJECTIVES

- Defining, Instantiating, and Starting Threads
- Preventing Thread Execution
- Synchronizing Code
- Thread Interaction

✓ Two-Minute Drill

Q&A Self Test

CERTIFICATION OBJECTIVE

Defining, Instantiating, and Starting Threads (Exam Objective 7.1)

Write code to define, instantiate, and start new threads using both java.lang.Thread and java.lang.Runnable.

Imagine a stockbroker application with a lot of complex behavior that the user initiates. One of the applications 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 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 that 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,

anytime 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*—as if in *parallel*—yet you know there's only one CPU on most of the machines running Java. 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 an understanding of how threads behave in different JVM environments required knowledge. In fact, the most important concept to understand from this entire chapter is

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.*

on the job

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 on the exam are among the most difficult. In fact, for most people they *are* the toughest questions on the exam, and with four 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 no attempt to teach you how to design a good, safe, multithreaded application!* You're here to learn 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 on the complexities and subtleties of multithreaded code.

With that out of the way, let's dive into threads. It's kind of a bad news/good news thing. The bad news is that this is probably the most difficult chapter. The good news is, *it's the last chapter* in the Programmer's Exam part of the book. So kick back and enjoy the fact that once you've finished learning what's in this chapter, and you've nailed the self-test questions, you're probably ready to take—and pass—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 all starts from 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 a good OO practice. Why? Because subclassing should be reserved for classes that extend an existing class, because they're a more specialized version of the more general superclass. 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 from 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 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 won't be called unless you call it.* It will not be used as the basis of a new call stack.

Implementing java.lang.Runnable

Implementing the Runnable interface gives you a way to extend from 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");
   }
}
```

Regardless of which mechanism you choose, you've now got yourself some code that can be run by a thread of execution. So now 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:

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

There are some additional overloaded constructors, but we'll look at those in a moment.

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.

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
```

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();
     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.

Besides the no-arg constructor and the constructor that takes a Runnable (the target, the instance with the job to do), there are other overloaded constructors in class Thread. The complete list of constructors is

- Thread()
- Thread(Runnable target)

- Thread(Runnable target, String name)
- Thread(String name)
- Thread(ThreadGroup group, Runnable target)
- Thread(ThreadGroup group, Runnable target, String name)
- Thread(ThreadGroup group, 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 to be *alive*. The "aliveness" of a thread can be tested by calling the `isAlive()` method on the Thread instance. In a nutshell, a thread is considered *alive* at some point after it has been started (you have to give the JVM a little time to get it set up as a thread once `start()` is called), and it is considered *not alive* after it becomes dead. The `isAlive()` method is the best way to determine if a thread has been started but has not yet completed its `run()` method.

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 subhead:

```
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.

exam Watch

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:

```
Runnable r = new Runnable();
r.run();  // Legal, but does not start a separate thread
```

The following example demonstrates what we've covered so far—defining, instantiating, and starting a thread:

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

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

```
% java TestThreads
Runnable running
Runnable running
Runnable running
```

```
Runnable running
Runnable running
```

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

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 name 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 name of 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 9-1 shows the process of starting a thread.

Starting and Running More Than One Thread

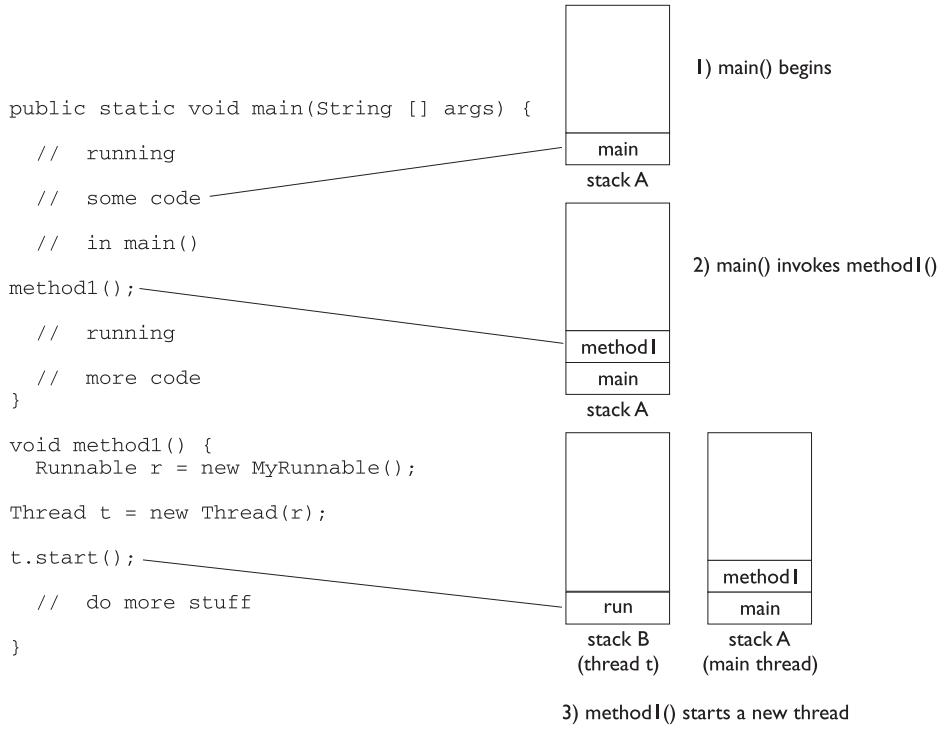
Enough playing around here; let's actually get *multiple* threads going. The following code creates a single Runnable instance, and three Thread instances. All three Thread instances get the same Runnable instance, and each thread is given a unique name. Finally, all three threads are started by invoking `start()` on the Thread instances.

```
class NameRunnable implements Runnable {
   public void run() {
      for (int x = 1; x < 4; x++) {
         System.out.println("Run by " + Thread.currentThread().getName());
      }
    }
```

```
}
public class ManyNames {
   public static void main (String [] args) {
     NameRunnable nr = new NameRunnable(); // Make one Runnable
     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();
   }
}
```

FIGURE 9-1

Starting a thread



Running this code produces the following:

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

Well, at least that's what it prints on *one* machine—the one we used for this particular example. (OK, if you insist we'll tell you—it's a Macintosh G4 Titanium running OSX. Yes Virginia, there *is* UNIX on the Mac.)

But the behavior you see above 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.

But how that happens is not just *JVM* dependent; it is also *runtime* dependent. In fact, just for fun we bumped up the loop code so that each `run()` method ran the loop 300 times rather than 3, and eventually we did start to see some wobbling:

```
public void run() {
      for (int x = 0; x < 300; x++) {
         System.out.println("Run by " + Thread.currentThread().getName());
      }
 }
```

Running the preceding code, with each thread executing its run loop 300 times, started out fine but then became nonlinear. Here's just a snip from the command-line

output of running that code. To make it easier to distinguish each thread, I put Fred's output in italics and Lucy's in bold, and left Ricky's alone:

```
Run by Fred
Run by Fred
Run by Fred
Run by Fred
Run by Fred
Run by Fred
Run by Fred
Run by Fred
Run by Fred
Run by Fred
Run by Lucy
Run by Ricky
Run by Fred
Run by Ricky
Run by Fred
Run by Ricky
Run by Fred
Run by Ricky
Run by Fred
Run by Ricky
... it continues on ...
```

Notice that Fred (who was started first) is humming along just fine for a while and then suddenly Lucy (started second) jumps in—but only runs once! She does finish later, of course, but not until after Fred and Ricky swap in and out with no clear pattern. The rest of the output also shows Lucy and Ricky swapping for a while, and then 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 its 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*. 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 is dead, it can never be restarted!

If you have a reference to a Thread *t*, and its `run()` method has finished, you can't say `t.start();` you'll get a big fat runtime exception.

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.

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 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.

exam Watch

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 indeterminate.

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()
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()
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. We've already seen three—*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 9-2):

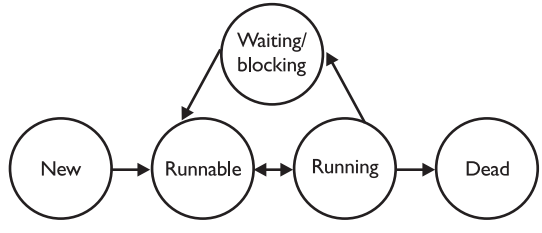
- **New** This is the state the thread is in after the Thread instance has been instantiated, but the `start()` method has not been invoked on the thread. It is a live Thread object, but not yet a thread of execution. At this point, the thread is considered *not alive*.
- **Runnable** This is the state 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 9-2, 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** OK, 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* 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 is that it wakes up because its sleep time has expired. Or the thread may be *waiting*, because the thread's run code *causes* it to wait, in which case the event that sends it back to runnable is that another thread sends a notification that it may no longer be necessary for the thread to wait. The important point is that one thread does not *tell* another thread to block. 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 to be *alive*.

FIGURE 9-2

Transitioning between thread states



- **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 a runtime (not compiler) exception. And it probably doesn't take a rocket scientist to tell you that if a thread is dead, it is no longer considered to be *alive*.

CERTIFICATION OBJECTIVE

Preventing Thread Execution (Exam Objective 7.2)

Recognize conditions that might prevent a thread from executing.

This objective has been the source of a lot of confusion over the last few years, because earlier versions of the objective weren't as clear about one thing: we're talking about moving a thread to a nonrunnable state (in other words, moving a thread to the blocked/sleeping/waiting state), as opposed to talking about what might *stop* a thread. A thread that's been stopped usually means a thread that's moved to the dead state. But Objective 7.2 is looking for your ability to recognize when a thread will get kicked out of running but not 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 specifications. 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 importantly—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 (which you'll usually know if that were a possibility, since another thread has to explicitly do the interrupting), so you're forced to acknowledge the exception with a handle or declare. Typically, you just wrap each call 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 `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) {
     NameRunnable nr = new NameRunnable(); // Make one Runnable
     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 there are two threads 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).

exam Watch

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 9-1

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 modulo operation to check whether there are any remainder numbers when divided by 10.
4. Use the static method `Thread.sleep()` to pause. The *long* number represents milliseconds.

Thread Priorities and Yield

To understand `yield()`, you must understand the concept of thread *priorities*. Threads always run with some priority, represented usually 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. *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, another 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 higher 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. *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 correct behavior of your program.

on the job

Don't rely on thread priorities when designing your multithreaded application. Because thread-scheduling priority behavior is not guaranteed, use thread priorities as a way to improve the efficiency of your program, but just be sure your program doesn't** depend **on that behavior for correctness.

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 keep it there until it blocks or completes its `run()` method.
- 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, since 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, the values 1 through 10 are not guaranteed, so if you have, say, ten threads each with a different priority, and the current application is running in a JVM that allocates a range of only five priorities, then two or more threads might be mapped to one priority. *The default priority is 5.*

The Thread class has three constants (static final variables) that define the range of thread priorities:

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

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.

The Join() Method

The nonstatic `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).

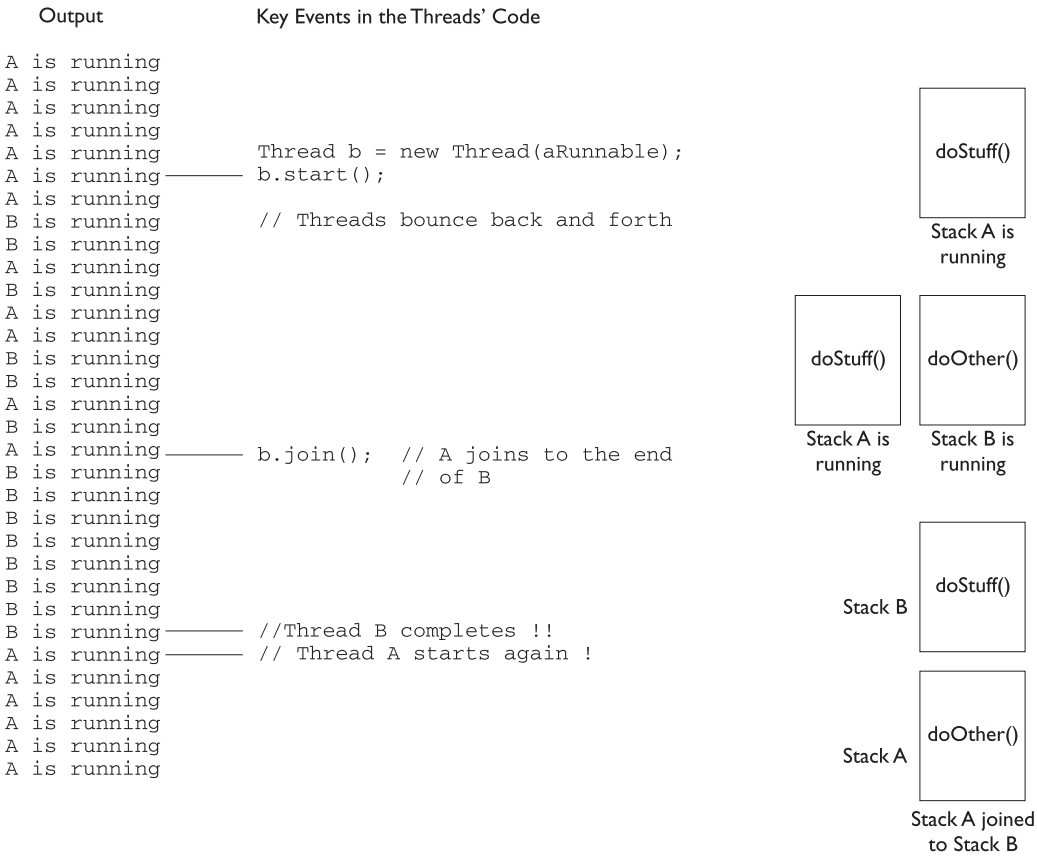
```
Thread t = new Thread();
t.start();
t.join();
```

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. 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 9-3 shows the effect of the `join()` method.

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).

FIGURE 9-3 The `join()` method



- 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 `wait()` on) completes. If the thread it's trying to join with is not alive, however, 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.

To understand the two critical execution stoppers, you need to understand the way in which Java implements object locking to prevent multiple threads from accessing—and potentially corrupting—the same data. Since these are covered in the next two objectives (7.3 and 7.4), be sure you study these so you'll recognize when a running thread will stop (at least temporarily) running.

CERTIFICATION OBJECTIVE

Synchronizing Code (Exam Objective 7.3)

Write code using synchronized wait, notify, and notifyAll to protect against concurrent access problems and to communicate between threads.

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 cover 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 that 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. So now imagine that Lucy makes *her* withdrawal, and now there isn't enough in the account for Fred's withdrawal, but he thinks there is since when he checked, there was enough! Yikes. Here's what the actual banking code looks like, 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 5 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) does 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.
   - When you wake up, complete the withdrawal and print out that you've done so.
   - 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 OK to withdraw, and the account will overdraw 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())
     }
   }
}
```

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 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 OK. At line 2, Lucy checks the balance and finds it OK. 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 its safe to do the withdrawal!* On line 11, Fred completes the withdrawal he approved on line 8. This takes the balance to zero. 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 9-4 shows the timeline of what can happen when two threads concurrently access the same object.

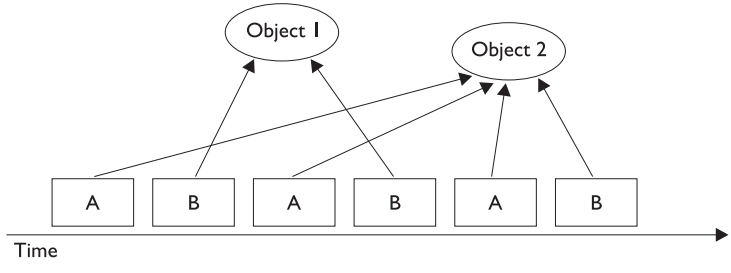
Preventing the Account Overdraw So what can be done? The solution is actually quite simple.

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 there is a little outdated) because the operation, regardless of the number of actual statements (or underlying byte code 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.

FIGURE 9-4

Problems with concurrent access



Thread A will access Object 2 only

Thread B will access Object 1, and then Object 2

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 that 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 the 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. Since there is only one lock per object, if one thread has picked up the lock, no other thread can enter the synchronized code (which means *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.

You need to remember the following key points about locking and synchronization:

- Only *methods* can be synchronized, not variables.
- Each object has just *one* lock.
- *Not all methods in a class must be synchronized.* A class can have both synchronized and nonsynchronized methods.
- If two methods are synchronized in a class, only one thread can be accessing one of the two methods. 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 nonsynchronized methods, *multiple threads can still access the nonsynchronized methods* of the class! If you have methods that don't access the data you're trying to protect, then you don't

need to mark them as synchronized. Synchronization is a performance hit, so you don't want to use it without a good reason.

- *If a thread goes to sleep, it takes its locks with it.*
- *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 example, 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.

So What About Static Methods? Can They 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 the static methods of the class (if they're synchronized).

There's nothing special you have to do to synchronize a static method:

```
public static synchronized void getCount() { }
```

EXERCISE 9-2

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 As, 100 Bs, and 100 Cs 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 threaded 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. You can check Chapter 11 for StringBuffer methods that will help with this.
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 an 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, however. 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.

Table 9-1 lists the thread-related methods and whether the thread gives up its locks as a result of the call.

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 it gives up the lock, so they'll sit there forever and ever and ever… 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—lock 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.

TABLE 9-1 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

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 forever waiting for the other to back down.

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 very 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: *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 then most Java programmers.

CERTIFICATION OBJECTIVE

Thread Interaction (Exam Objective 7.4)

Define the interaction among threads and object locks when executing synchronized wait, notify, and notifyAll.

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 java.lang.Object class has three methods—`wait()`, `notify()`, and `notifyAll()`—that help threads communicate about 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 over here, 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, 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 and 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 only one thread, which first asks the user for instructions and then directs the hardware to cut that shape, repeating the cycle afterward.

```
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.        }
12.        System.out.println("Total is: " + b.total);
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 12 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 is common to find code such as the following:

```
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 any 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 fabric 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); // The thread releases the lock and waits for notify
   // But only for a maximum of two seconds, then goes back to Runnable
   // The thread reacquires the lock
   // More instructions here
}
```

exam Watch

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 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 threads waiting on one object, but for different reasons, and you want to be sure that the *right* thread gets 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.        }
15.        System.out.println("Total is: " + c.total);
16.     }
17.
18.     public static void main(String [] args) {
19.        Calculator calculator = new Calculator();
20.        calculator.start();
21.        new Reader(calculator).start();
22.        new Reader(calculator).start();
23.        new Reader(calculator).start();
24.     }
25.  }
26.
```

```
27.  class Calculator extends Thread {
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.  }
```

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

exam Watch

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 9-2 lists the key methods you'll need to know for the exam, with the static methods shown in italics.

TABLE 9-2 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 overridden 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. To coordinate activity between different threads, use the `wait()`, `notify()`, and `notifyAll()` methods.

TWO-MINUTE DRILL

Here are some of the key points from each certification objective in Chapter 9. Photocopy it and sleep with it under your pillow for complete absorption.

Creating, Instantiating, and Starting New Threads

- ❑ 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.
- ❑ 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 RuntimeException.
- ❑ 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

- ❑ 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.
- ❑ 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 should can use the `sleep()` method. This prevents one thread from hogging the running process while another thread starves.

- ❑ 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 running (well, for all practical purposes anyway).
- ❑ A dead thread cannot be started again.

Sleep, Yield, and Join

- ❑ 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. One thread *cannot* tell another thread to sleep.
- ❑ The `setPriority()` method is used on Thread objects to give threads a priority of between 1 (low) and 10 (high), although priorities are not guaranteed, and not all JVMs use a priority range of 1-10.
- ❑ If not explicitly set, a thread's priority will be the same priority as the thread that created this thread (in other words, the thread executing the code that creates the new thread).
- ❑ 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 it will be *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 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

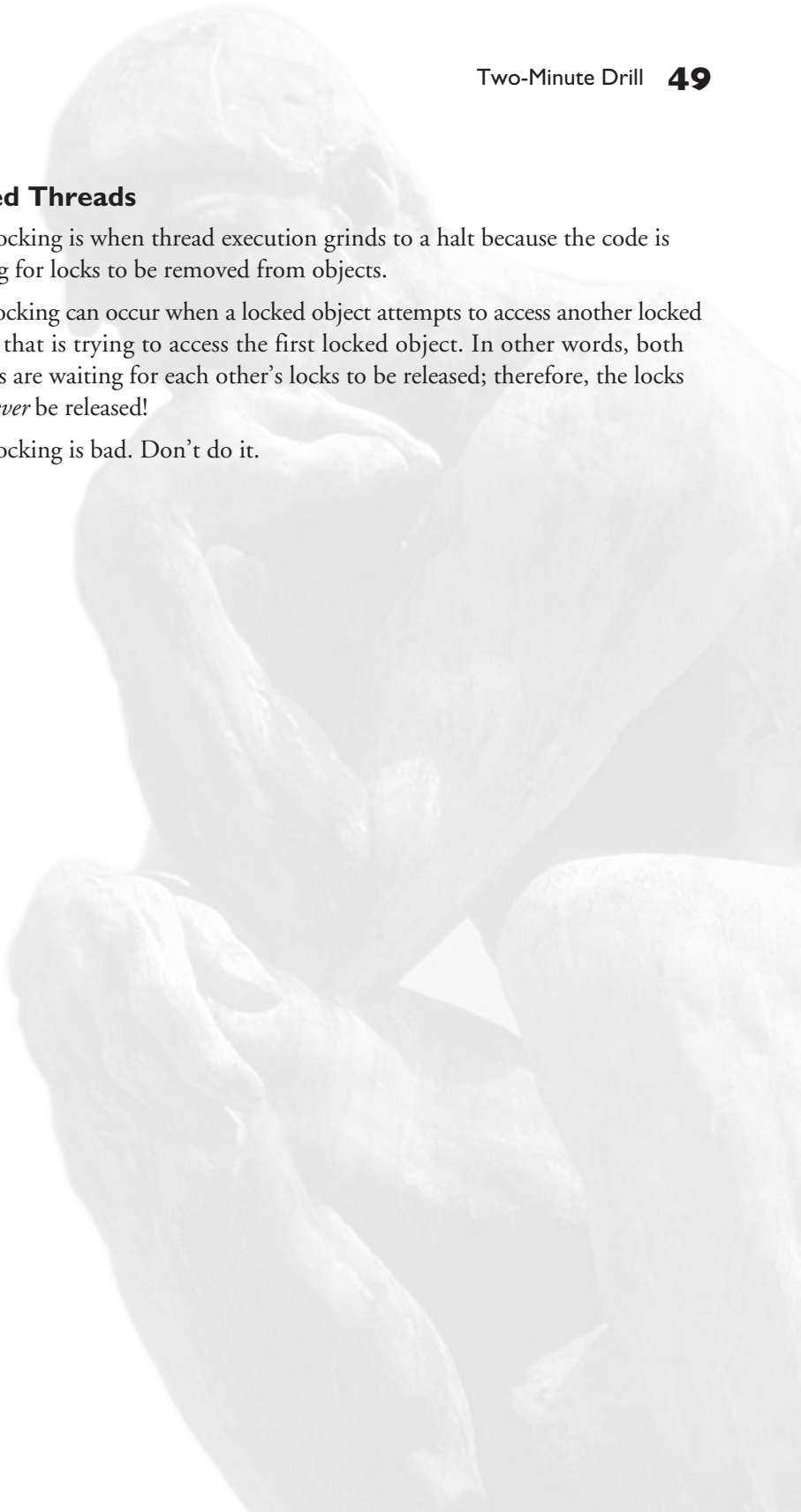
- Synchronized methods prevent more than one thread from accessing an object's critical method code.
- You can use the `synchronized` keyword as a method modifier, or to start a synchronized block of code.
- To synchronize a block of code (in other words, a scope smaller than the whole method), you must 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 *un*synchronized code.
- When an object goes to sleep, it takes its locks with it.
- Static methods can be synchronized, using the lock from the java.lang.Class instance representing that class.

Communicating with Objects by Waiting and Notifying

- The `wait()` method lets a thread say, "there's nothing for me to do here, so put me in your waiting pool and notify me when something happens that I care about." Basically, a `wait()` call means "wait me 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 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()`/`notifyAll()`—must be called from within a synchronized context! A thread invokes `wait()`/`notify()` on a particular object, and the thread must currently hold the lock on that object.

Deadlocked Threads

- ❑ 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.



SELF TEST

The following questions will help you measure your understanding of the material presented in this chapter. Read all of the choices carefully, as there may be more than one correct answer. Choose all correct answers for each question. Stay focused.

If you have a rough time with these at first, don't beat yourself up. Be positive. Repeat nice affirmations to yourself like, "I am smart enough to understand threads." "I can do this." and "OK, so that other guy knows threads better than I do, but I bet he can't <insert something you *are* good at> like me."

1. Given the following,

```
1.  class MyThread extends Thread {
2.
3.     public static void main(String [] args) {
4.        MyThread t = new MyThread();
5.        t.run();
6.     }
7.
8.     public void run() {
9.        for(int i=1;i<3;++i) {
10.          System.out.print(i + "..");
11.       }
12.    }
13. }
```

what is the result?

A. This code will not compile due to line 4.

B. This code will not compile due to line 5.

C. 1..2..

D. 1..2..3..

E. An exception is thrown at runtime.

2. Which two of the following methods are defined in class Thread?

A. `start()`

B. `wait()`

C. `notify()`

D. `run()`

E. `terminate()`

3. 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(){}}`

4. Given the following,

```
1.  class MyThread extends Thread {
2.
3.     public static void main(String [] args) {
4.        MyThread t = new MyThread();
5.        t.start();
6.        System.out.print("one. ");
7.        t.start();
8.        System.out.print("two. ");
9.     }
10.
11.    public void run() {
12.       System.out.print("Thread ");
13.    }
14. }
```

what is the result of this code?

A. Compilation fails

B. An exception occurs at runtime.

C. `Thread one. Thread two.`

D. The output cannot be determined.

5. Given the following,

```
1.  public class MyRunnable implements Runnable {
2.     public void run() {
3.        // some code here
4.     }
5.  }
```

which of these will create and start this thread?

A. `new Runnable(MyRunnable).start();`

B. `new Thread(MyRunnable).run();`

C. `new Thread(new MyRunnable()).start();`

D. `new MyRunnable().start();`

6. Given the following,

```
1.  class MyThread extends Thread {
2.
3.     public static void main(String [] args) {
4.        MyThread t = new MyThread();
5.        Thread x = new Thread(t);
6.        x.start();
7.     }
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.

7. Given the following,

```
1.  class Test {
2.
3.     public static void main(String [] args) {
4.        printAll(args);
5.     }
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 1-second pause.

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 there will be a pause because `currentThread()` may not retrieve this thread.

D. This code will not compile.

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

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

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

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

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

9. Which class or interface defines the `wait()`, `notify()`, and `notifyAll()` methods?

A. Object

B. Thread

C. Runnable

D. Class

10. Which two are *true*?

A. A static method cannot be synchronized.

B. If a class has synchronized code, multiple threads can still access the nonsynchronized code.

C. Variables can be protected from concurrent access problems by marking them with the `synchronized` keyword.

D. When a thread sleeps, it releases its locks.

E. When a thread invokes `wait()`, it releases its locks.

11. Which three are methods of the Object class? (Choose three.)

A. `notify();`

B. `notifyAll();`

C. `isInterrupted();`

D. `synchronized();`

E. `interrupt();`

F. `wait(long msecs);`

G. `sleep(long msecs);`

H. `yield();`

12. Given the following,

```
 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.

13. 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 the thread A become a candidate to get another turn at the CPU?

A. After thread A is notified, or after two seconds.
B. After the lock on B is released, or after two seconds.
C. Two seconds after thread A is notified.
D. Two seconds after lock B is released.

14. Which two are *true*?

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 release its locks.
E. The `notify()` method causes a thread to immediately release its locks.
F. The difference between `notify()` and `notifyAll()` is that `notifyAll()` notifies all waiting threads, regardless of the object they're waiting on.

15. Assume you create a program and one of your threads (called backgroundThread) does some lengthy numerical processing. What would be the proper way of setting its priority to try to get the rest of the system to be very responsive while the thread is running? (Choose all that apply.)

A. `backgroundThread.setPriority(Thread.LOW_PRIORITY);`
B. `backgroundThread.setPriority(Thread.MAX_PRIORITY);`
C. `backgroundThread.setPriority(1);`
D. `backgroundThread.setPriority(Thread.NO_PRIORITY);`
E. `backgroundThread.setPriority(Thread.MIN_PRIORITY);`
F. `backgroundThread.setPriority(Thread.NORM_PRIORITY);`
G. `backgroundThread.setPriority(10);`

16. Which three guarantee that a thread will leave the running state?

A. `yield()`
B. `wait()`
C. `notify()`
D. `notifyAll()`

E. `sleep(1000)`

F. `aLiveThread.join()`

G. `Thread.killThread()`

17. Which two are true?

A. Deadlock will not occur if `wait()/notify()` is used.

B. A thread will resume execution as soon as its sleep duration expires.

C. Synchronization can prevent two objects from being accessed by the same thread.

D. The `wait()` method is overloaded to accept a duration.

E. The `notify()` method is overloaded to accept a duration.

F. Both `wait()` and `notify()` must be called from a synchronized context.

G. `wait()` can throw a runtime exception

H. `sleep()` can throw a runtime exception.

18. Which two are valid constructors for Thread?

A. `Thread(Runnable r, String name)`

B. `Thread()`

C. `Thread(int priority)`

D. `Thread(Runnable r, ThreadGroup g)`

E. `Thread(Runnable r, int priority)`

19. Given the following,

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

```
        System.out.println(" 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.

20. Given the following,

```
public class SyncTest {
   public static void main (String [] args) {
     Thread t = new Thread() {
       Foo f = new Foo();
       public void run() {
         f.increase(20);
       }
     };
     t.start();
   }
}
class Foo {
  private int data = 23;
   public void increase(int amt) {
       int x = data;
       data = x + amt;
    }
}
```

and assuming that data must be protected from corruption, what—if anything—can you *add* to the preceding code to ensure the integrity of data?

A. Synchronize the run method.
B. Wrap a synchronize(*this*) around the call to `f.increase()`.

C. The existing code will not compile.

D. The existing code will cause a runtime exception.

E. Put in a `wait()` call prior to invoking the `increase()` method.

F. Synchronize the `increase()` method

21. Given the following,

```
 1. public class Test {
 2.  public static void main (String [] args) {
 3.    final Foo f = new Foo();
 4.   Thread t = new Thread(new Runnable() {
 5.         public void run() {
 6.        f.doStuff();
 7.       }
 8.   });
 9.   Thread g = new Thread() {
10.    public void run() {
11.         f.doStuff();
12.      }
13.     };
14.     t.start();
15.     g.start();
16.  }
17. }
 1. class Foo {
 2.   int x = 5;
 3.  public void doStuff() {
 4.     if (x < 10) {
 5.       // nothing to do
 6.      try {
 7.         wait();
 8.      } catch(InterruptedException ex) { }
 9.     } else {
10.      System.out.println("x is " + x++);
11.      if (x >= 10) {
12.          notify();
13.        }
14.    }
15.  }
16. }
```

what is the result?

A. The code will not compile because of an error on line 12 of class Foo.
B. The code will not compile because of an error on line 7 of class Foo.
C. The code will not compile because of an error on line 4 of class Test.
D. The code will not compile because of some other error in class Test.
E. An exception occurs at runtime.
F. *x* is 5
   x is 6

SELF TEST ANSWERS

1. ☑ **C.** Line 5 calls the `run()` method, so the `run()` method executes as a normal method should.
   ☒ **A** is incorrect because line 4 is the proper way to create an object. **B** is incorrect because it is legal to call the `run()` method, even though this will not start a true thread of execution. The code after line 5 will not execute until the `run()` method is complete. **D** is incorrect because the *for* loop only does two iterations. **E** is incorrect because the program runs without exception.

2. ☑ **A** and **D.** Only `start()` and `run()` are defined by the Thread class.
   ☒ **B** and **C** are incorrect because they are methods of the Object class. **E** is incorrect because there's no such method in any thread-related class.

3. ☑ **C.** The class correctly implements the Runnable interface with a legal `public void run()` method.
   ☒ **A** is incorrect because interfaces are not extended; they are implemented. **B** is incorrect because even though the class would compile and it has a valid `public void run()` method, it does not implement the Runnable interface, so the compiler would complain when creating a Thread with an instance of it. **D** is incorrect because the `run()` method must be *public*. **E** is incorrect because the method to implement is `run()`, not `start()`.

4. ☑ **B.** When the `start()` method is attempted a second time on a single Thread object, the method will throw an IllegalThreadStateException (you will not need to know this exception name for the exam). Even if the thread has finished running, it is still illegal to call `start()` again.
   ☒ **A** is incorrect because compilation will succeed. For the most part, the Java compiler only checks for illegal syntax, rather than class-specific logic. **C** and **D** are incorrect because of the logic explained above.

5. ☑ **C.** Because the class implements Runnable, an instance of it has to be passed to the Thread constructor, and then the instance of the Thread has to be started.
   ☒ **A** is incorrect. There is no constructor like this for Runnable because Runnable is an interface, and it is illegal to pass a class or interface name to any constructor. **B** is incorrect for the same reason; you can't pass a class or interface name to any constructor. **D** is incorrect because MyRunnable doesn't have a `start()` method, and the only `start()` method that can start a thread of execution is the `start()` in the Thread class.

6. ☑ **D.** 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 5, Thread can take an object of type Thread as an argument in the constructor. **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 because of the program logic described above.

7. ☑ **D.** The `sleep()` method must be enclosed in a `try/catch` block, or the method `printAll()` must declare it `throws` the InterruptedException.
☒ **A** is incorrect, but it would be correct if the InterruptedException was dealt with. **B** is incorrect, but it would still be incorrect 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, so even if it is called on an instance, it still always affects the currently executing thread.

8. ☑ **B** and **F.** By marking the methods as `synchronized`, the threads will get the lock of the *this* object before proceeding. Only one thread will be either 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 is not possible to select other objects to synchronize on when declaring a method as `synchronized`. Even using *this* is incorrect syntax.

9. ☑ **A.** The Object class defines these thread-specific methods.
☒ **B, C,** and **D** are incorrect because they do not define these methods. And yes, the Java API does define a class called Class, though you do not need to know it for the exam.

10. ☑ **B** and **E. B** is correct because multiple threads are allowed to enter nonsynchronized code, even within a class that has some synchronized methods. **E** is correct because a `wait()` call causes the thread to give up its locks.
☒ **A** is incorrect because static methods can be synchronized; they synchronize on the lock on the instance of class java.lang.Class that represents the class type. **C** is incorrect because only methods—not variables—can be marked `synchronized`. **D** is incorrect because a sleeping thread still maintains its locks.

11. ☑ **A, B,** and **F.** They are all related to the list of threads waiting on the specified object.
☒ **C, E, G,** and **H** are incorrect answers. The methods `isInterrupted()` and `interrupt()` are instance methods of Thread. The methods `sleep()` and `yield()` are static methods of Thread. **D** is incorrect because `synchronized` is a keyword and the `synchronized()` construct is part of the Java language.

12. ☑ **D.** 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. The program is essentially frozen at line 7.

☒ **A** is incorrect; IllegalMonitorStateException is an unchecked exception so it doesn't have to be dealt with explicitly. **B** and **C** are incorrect; 3 will never be printed, since this program will never terminate because it 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, furthermore, there is no *this* when running a static method.

13. ☑ **A.** Either of the two events (notification or wait time expiration) will make the thread become 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, not two seconds afterwards. **D** is also incorrect because a thread will not come out of a waiting pool just because a lock has been released.

14. ☑ **A** and **D.** A is correct because the `notifyAll()` method (along with `wait()` and `notify()`) must always be called from within a synchronized context. **D** is correct because a thread blocked on a `wait()` call releases its locks, so another thread can get into the synchronized code and eventually call `notify()` or `notifyAll()`.

☒ **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 wrong because `notify()` is defined in java.lang.Object. **E** is wrong 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 wrong because `notifyAll()` notifies all the threads waiting on a particular locked object, not all threads waiting on *any* object.

15. ☑ **C** and **E.** In **E**, the constant Thread.MIN_PRIORITY is the lowest priority that a thread can have, and the background thread should have a very low priority or the lowest. Answer **C** is correct because 1 is a low (and usually the minimum) value, although for code clarity it is recommended to use the Thread.MIN_PRIORITY.

☒ **A** and **D** are incorrect because there are no such variables in the Thread class. **B** is incorrect; using MAX_PRIORITY would make other threads have fewer chances of getting a turn of the CPU, even to the point of freezing until the numerical processing is finished. **F** is incorrect because the thread would still compete for the CPU time and even delay other threads. **G** is incorrect because 10 is the value of MAX_PRIORITY, so *i* would be equivalent to answer **B**.

16. ☑ **B**, **E**, and **F**. **B** is correct because `wait()` always causes the current thread to go into the object's wait pool. **E** is correct because `sleep()` will always pause the currently running thread for *at least* the duration specified in the sleep argument (unless an interrupted exception is thrown). **F** is correct because, assuming that the thread you're calling `join()` on is alive, the thread calling `join()` will immediately block until the thread you're calling `join()` on is no longer alive.

☒ **A** is wrong, but tempting. The `yield()` method is not guaranteed to cause a thread to leave the running state, although if there are runnable threads of the same priority as the currently running thread, then the current thread will *probably* leave the running state. **C** and **D** are incorrect because they don't cause the thread invoking them to leave the running state. **G** is wrong because there's no such method.

17. ☑ **D** and **F**. **D** is correct because the `wait()` method is overloaded to accept a wait duration in milliseconds. If the thread has not been notified by the time the wait duration has elapsed, then the thread will move back to runnable even *without* having been notified. **F** is correct because `wait()`/`notify()`/`notifyAll()` must all be called from within a synchronized, context. A thread must own the lock on the object its invoking `wait()`/`notify()`/`notifyAll()` on.

☒ **A** is incorrect because `wait()`/`notify()` will not prevent deadlock. **B** is incorrect because a sleeping thread will return to runnable when it wakes up, but it might not necessarily resume execution right away. To resume executing, the newly awakened thread must still be moved from runnable to running by the scheduler. **C** is incorrect because synchronization prevents two or more threads from accessing the same object. **E** is incorrect because `notify()` is not overloaded to accept a duration. **G** and **H** are incorrect because `wait()` and `sleep()` both declare a checked exception (InterruptedException).

18. ☑ **A** and **B** are both valid constructors for Thread.

☒ **C**, **D**, and **E** are not legal Thread constructors, although **D** is close. If you reverse the arguments in **D**, you'd have a valid constructor.

19. ☑ **B** is correct because 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". The next statement in main invokes `start()` on the new thread instance, which causes the overridden `run()` method (the `run()` method defined in the anonymous inner class) to be invoked, which prints " foo".

☒ **A**, **C**, **D**, **E**, **F**, and **G** are all incorrect because of the program logic described above.

20. ☑ **F** is correct because synchronizing the code that actually does the increase will protect the code from being accessed by more than one thread at a time.
☒ **A** is incorrect because synchronizing the `run()` method would stop other threads from running the `run()` method (a bad idea) but still would not prevent other threads with *other* runnables from accessing the `increase()` method. **B** is incorrect for virtually the same reason as **A**—synchronizing the code that *calls* the `increase()` method does not prevent other code from calling the `increase()` method. **C** and **D** are incorrect because the program compiles and runs fine. **E** is incorrect because it will simply prevent the call to `increase()` from ever happening from this thread.

21. ☑ **E** is correct because the thread does not own the lock of the object it invokes `wait()` on. If the method were synchronized, the code would run without exception.
☒ **A, B, C,** and **D** are incorrect because the code compiles without errors. **F** is incorrect because the exception is thrown before there is any output.

EXERCISE ANSWERS

Exercise 9-1: 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 9-2: Synchronizing a Block of Code

Your code might look something like this when completed:

```
 1.  class InSync extends Thread {
 2.     StringBuffer letter;
 3.
 4.     public InSync(StringBuffer letter) {
 5.        this.letter = letter;
 6.     }
 7.
 8.     public void run() {
 9.        synchronized(letter) {
10.           for(int i = 1;i<=100;++i) {
11.              System.out.print(letter);
12.           }
13.           System.out.println();
14.           // Increment the letter in StringBuffer:
15.           char temp = letter.charAt(0);
16.           ++temp;
17.           letter.setCharAt(0, temp);
18.        }
19.     }
20.
21.     public static void main(String [] args) {
22.        StringBuffer sb = new StringBuffer("A");
23.        new InSync(sb).start();
24.        new InSync(sb).start();
25.        new InSync(sb).start();
26.     }
27.  }
```

Just for fun, try removing lines 9 and 18 then run the program again. It will be unsynchronized, and watch what happens.