



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

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

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

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

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

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

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

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

Thread States and Transitions

We've already seen three thread states—*new*, *runnable*, and *dead*—but wait! There's more! The thread scheduler's job is to move threads in and out of the

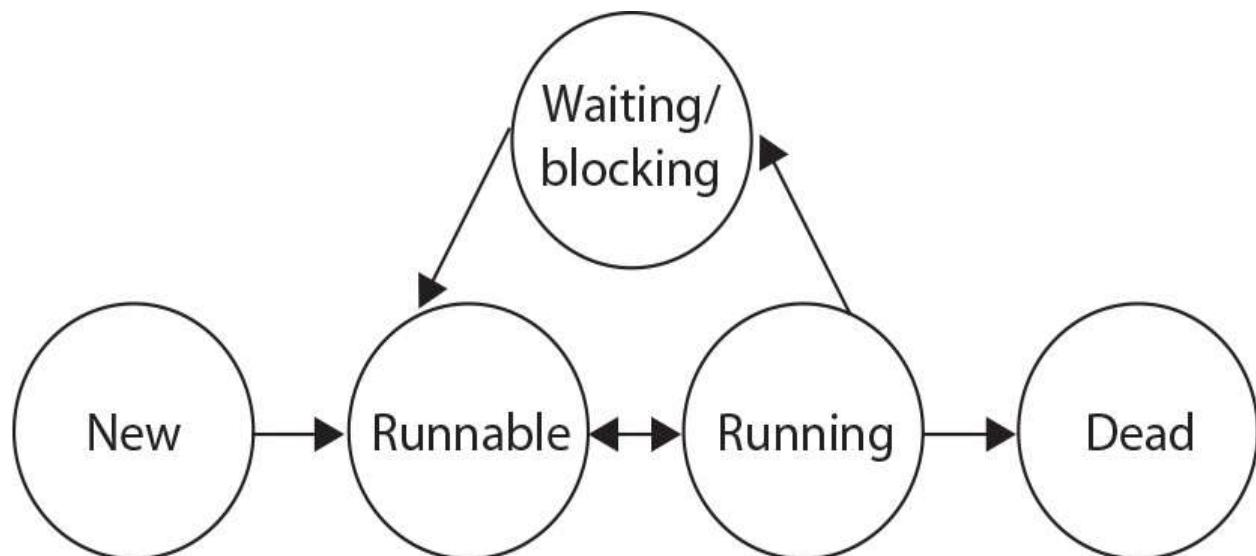
running state. While the thread scheduler can move a thread from the running state back to runnable, other factors can cause a thread to move out of running, but *not* back to runnable. One of these is when the thread’s `run()` method completes, in which case, the thread moves from the running state directly to the dead state. Next, we’ll look at some of the other ways in which a thread can leave the running state and where the thread goes.

Thread States

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

FIGURE 10-2

Transitioning between thread states



- **New** This is the state the thread is in after the `Thread` instance has been created but the `start()` method has not been invoked on the thread. It is a live `Thread` object, but not yet a thread of execution. At this point, the thread is considered *not alive*.
- **Runnable** This is the state a thread is in when it’s eligible to run but the scheduler has not selected it to be the running thread. A thread first enters the runnable state when the `start()` method is invoked, but a thread can also return to the runnable state after either running or coming back from a blocked, waiting, or sleeping state. When the thread is in the runnable state, it is considered *alive*.
- **Running** This is it. The “big time.” Where the action is. This is the state

a thread is in when the thread scheduler selects it from the runnable pool to be the currently executing process. A thread can transition out of a running state for several reasons, including because “the thread scheduler felt like it.” We’ll look at those other reasons shortly. Note that in [Figure 10-2](#), there are several ways to get to the runnable state, but only *one* way to get to the running state: the scheduler chooses a thread from the runnable pool.

- **Waiting/blocked/sleeping** This is the state a thread is in when it’s not eligible to run. Okay, so this is really three states combined into one, but they all have one thing in common: the thread is still alive but is currently not eligible to run. In other words, it is not *Runnable*, but it might *return* to a runnable state later if a particular event occurs. A thread may be *blocked* because it’s waiting for a resource (like I/O or an object’s lock), in which case the event that sends it back to runnable is the availability of the resource—for example, if data comes in through the input stream the thread code is reading from or if the object’s lock suddenly becomes available. A thread may be *sleeping* because the thread’s run code *tells* it to sleep for some period of time, in which case, the event that sends it back to runnable causes it to wake up because its sleep time has expired. Or the thread may be *waiting* because the thread’s run code *causes* it to wait. In that case, an event occurs, causing another thread to be sent a notification that it may no longer be necessary to wait. Then the waiting thread will become runnable again. The important point is that one thread does not *tell* another thread to block. Some methods may *look* like they tell another thread to block, but they don’t. If you have a reference *t* to another thread, you can write something like this:

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

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

a thread in a blocked state is still considered *alive*.

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

Preventing Thread Execution

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

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

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

Sleeping

The `sleep()` method is a static method of class `Thread`. You use it in your code to “slow a thread down” by forcing it to go into a sleep mode before coming back to runnable (where it still has to beg to be the currently running thread). When a thread sleeps, it drifts off somewhere and doesn’t return to runnable until it wakes up.

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

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

milliseconds as follows:

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

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

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

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

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

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

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

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

which case, it immediately throws the `InterruptedException`).



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

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

EXERCISE 10-1

Creating a Thread and Putting It to Sleep

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

1. Create a class and extend the `Thread` class. As an option, you can implement the `Runnable` interface.
2. Override the `run()` method of `Thread`. This is where the code will go that will output the numbers.
3. Create a `for` loop that will loop 100 times. Use the modulus operation to check whether there are any remainder numbers when divided by 10.

4. Use the static method `Thread.sleep()` to pause. (Remember, the one-arg version of `sleep()` specifies the amount of time of sleep in milliseconds.)
-

Thread Priorities and `yield()`

To understand `yield()`, you must understand the concept of thread *priorities*. Threads always run with some priority, usually represented as a number between 1 and 10 (although in some cases, the range is less than 10). The scheduler in most JVMs uses preemptive, priority-based scheduling (which implies some sort of time slicing). *This does not mean that all JVMs use time slicing.* The JVM specification does not require a VM to implement a time-slicing scheduler, where each thread is allocated a fair amount of time and then sent back to runnable to give another thread a chance. Although many JVMs do use time slicing, some may use a scheduler that lets one thread stay running until the thread completes its `run()` method.

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



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

What is also *not* guaranteed is the behavior when threads in the pool are of equal priority or when the currently running thread has the same priority as

threads in the pool. All priorities being equal, a JVM implementation of the scheduler is free to do just about anything it likes. That means a scheduler might do one of the following (among other things):

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

Setting a Thread's Priority

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

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

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

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

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

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

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

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

The `yield()` Method

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

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

The `join()` Method

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

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

The preceding code takes the currently running thread (if this were in the `main()` method, then that would be the main thread) and *joins* it to the end of the thread referenced by `t`. This blocks the current thread from becoming runnable

until after the thread referenced by `t` is no longer alive. In other words, the code `t.join()` means “Join me (the current thread) to the end of `t`, so that `t` must finish before I (the current thread) can run again.” You can also call one of the overloaded versions of `join()` that takes a timeout duration so that you’re saying, “Wait until thread `t` is done, but if it takes longer than 5,000 milliseconds, then stop waiting and become runnable anyway.” [Figure 10-3](#) shows the effect of the `join()` method.

FIGURE 10-3

The `join()` method

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

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

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

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

CERTIFICATION OBJECTIVE

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

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

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

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

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

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

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

Now here's where it starts to get fun. Imagine a couple, Fred and Lucy, who both have access to the account and want to make withdrawals. But they don't want the account to ever be overdrawn, so just before one of them makes a withdrawal, he or she will first check the balance to be certain there's enough to 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 there's just exactly enough in the account, 10. *But before she makes the withdrawal, Fred checks the balance and also sees that there's enough for his withdrawal.* Since Lucy has verified the balance but not yet made her withdrawal, Fred is seeing "bad data." He is seeing the account balance *before* Lucy actually debits the account, but at this point, that debit is certain to occur. Now both Lucy and Fred believe there's enough to make their withdrawals. Now imagine that Lucy makes *her* withdrawal, so there isn't enough in the account for Fred's withdrawal, but he thinks there is because when he checked, there was enough! Yikes. In a minute, we'll see the actual banking code, with Fred and Lucy, represented by two threads, each acting on the same `Runnable`, and that `Runnable` holds a reference to the one and only account instance—so, two threads, one account.

The logic in our code example is as follows:

1. The `Runnable` object holds a reference to a single account.
2. Two threads are started, representing Lucy and Fred, and each thread is given a reference to the same `Runnable` (which holds a reference to the actual account).
3. The initial balance on the account is 50, and each withdrawal is exactly 10.
4. In the `run()` method, we loop five times, and in each loop we
 - Make a withdrawal (if there's enough in the account).
 - Print a statement *if the account is overdrawn* (which it should never be since we check the balance *before* making a withdrawal).
5. The `makeWithdrawal()` method in the test class (representing the behavior of Fred or Lucy) will do the following:
 - Check the balance to see if there's enough for the withdrawal.
 - If there is enough, print out the name of the one making the

withdrawal.

- Go to sleep for 500 milliseconds—just long enough to give the other partner a chance to get in before you actually *make* the withdrawal.
- Upon waking up, complete the withdrawal and print that fact.
- If there wasn't enough in the first place, print a statement showing who you are and the fact that there wasn't enough.

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

Here's the code:

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

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

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

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

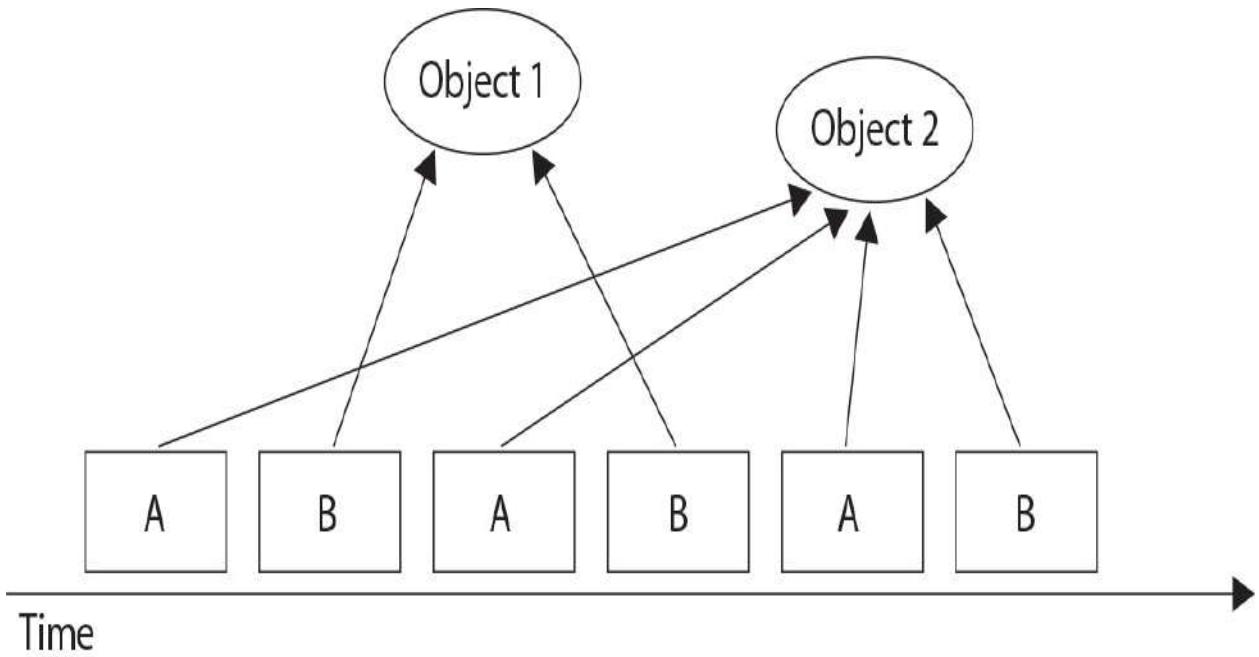
On line 5, Lucy completes her withdrawal and then, before Fred completes his, Lucy does another check on the account on line 6. And so it continues until we get to line 8, where Fred checks the balance and sees that it's 20. On line 9, Lucy completes a withdrawal that she had checked for earlier, and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

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

[Figure 10-4](#) shows the timeline of what can happen when two threads concurrently access the same object.

FIGURE 10-4

Problems with concurrent access



Thread A will access Object 2 only

Thread B will access Object 1, and then Object 2

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

Preventing the Account Overdraw

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

You can’t guarantee that a single thread will stay running throughout the entire atomic operation. But you can guarantee that even if the thread running the atomic operation moves in and out of the running state, no other running

thread will be able to act on the same data. In other words, if Lucy falls asleep after checking the balance, we can stop Fred from checking the balance until after Lucy wakes up and completes her withdrawal.

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

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

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

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

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

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

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

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

Synchronization and Locks

How does synchronization work? With locks. Every object in Java has a built-in lock that only comes into play when the object has synchronized method code.

When we enter a synchronized non-static method, we automatically acquire the lock associated with the current instance of the class whose code we're executing (the `this` instance). Acquiring a lock for an object is also known as getting the lock, or locking the object, locking *on* the object, or synchronizing on the object. We may also use the term *monitor* to refer to the object whose lock we're acquiring. Technically, the lock and the monitor are two different things, but most people talk about the two interchangeably, and we will too.

Since there is only one lock per object, if one thread has picked up the lock, no other thread can pick up the lock until the first thread releases (or returns) the lock. This means no other thread can enter the synchronized code (which means it can't enter any synchronized method of that object) until the lock has been released. Typically, releasing a lock means the thread holding the lock (in other words, the thread currently in the synchronized method) exits the synchronized method. At that point, the lock is free until some other thread enters a synchronized method on that object. Remember the following key points about locking and synchronization:

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

a synchronized method, thus acquiring a lock, and then immediately invoke a synchronized method on a different object, thus acquiring that lock as well. As the stack unwinds, locks are released again. Also, if a thread acquires a lock and then attempts to call a synchronized method on that same object, no problem. The JVM knows that this thread already has the lock for this object, so the thread is free to call other synchronized methods on the same object, using the lock the thread already has.

- You can synchronize a block of code rather than a method.

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

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

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

When you synchronize a method, the object used to invoke the method is the object whose lock must be acquired. But when you synchronize a block of code, you specify which object's lock you want to use as the lock, so you could, for 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.

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

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

is equivalent to this:

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

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

Can Static Methods Be Synchronized?

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

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

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

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

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

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

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

EXERCISE 10-2

Synchronizing a Block of Code

In this exercise, we will attempt to synchronize a block of code. Within that block of code, we will get the lock on an object so that other threads cannot

modify it while the block of code is executing. We will be creating three threads that will all attempt to manipulate the same object. Each thread will output a single letter 100 times and then increment that letter by one. The object we will be using is `StringBuffer`.

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

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

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

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

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

- Threads calling non-static synchronized methods in the same class will only block each other if they're invoked using the same instance. That's because they each lock on this instance, and if they're called

using two different instances, they get two locks, which do not interfere with each other.

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

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

So When Do I Need to Synchronize?

TABLE 10-1

Methods and Lock Status

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

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

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

want to; it doesn't matter.

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

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

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

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

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

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

Thread-Safe Classes

When a class has been carefully synchronized to protect its data (using the rules just given or using more complicated alternatives), we say the class is “thread-safe.” Many classes in the Java APIs already use synchronization internally in order to make the class “thread-safe.” For example, `StringBuffer` and `StringBuilder` are nearly identical classes, except that all the methods in

`StringBuffer` are synchronized when necessary, whereas those in `StringBuilder` are not. Generally, this makes `StringBuffer` safe to use in a multithreaded environment, whereas `StringBuilder` is not. (In return, `StringBuilder` is a little bit faster because it doesn't bother synchronizing.) However, even when a class is “thread-safe,” it is often dangerous to rely on these classes to provide the thread protection you need. (C'mon, the repeated quotes used around “thread-safe” had to be a clue, right?) You still need to think carefully about how you use these classes. As an example, consider the following class:

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

The method `Collections.synchronizedList()` returns a `List` whose methods are all synchronized and “thread-safe” according to the documentation (like a `Vector`—but since this is the 21st century, we’re not going to use a `Vector` here). The question is, can the `NameList` class be used safely from multiple threads? It’s tempting to think that yes, because the data in `names` is in a synchronized collection, the `NameList` class is “safe” too. However, that’s not the case—the `removeFirst()` may sometimes throw an

`IndexOutOfBoundsException`. What's the problem? Doesn't it correctly check the `size()` of `names` before removing anything to make sure there's something there? How could this code fail? Let's try to use `NameList` like this:

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

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

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

The output here is

```
Ozymandias
null
```

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

Thread t1 executes `names.size()`, which returns 1.

Thread t2 executes `names.size()`, which returns 1.

Thread t1 executes `names.remove(0)`, which returns Ozymandias.

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

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

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

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

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

completes the `removeFirst()` method.

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

Thread Deadlock

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

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

```
1. public class DeadlockRisk {  
2.     private static class Resource {  
3.         public int value;  
4.     }  
5.     private Resource resourceA = new Resource();  
6.     private Resource resourceB = new Resource();  
7.     public int read() {  
8.         synchronized(resourceA) { // May deadlock here  
9.             synchronized(resourceB) {  
10.                 return resourceB.value + resourceA.value;  
11.             }  
12.         }  
13.     }  
14.  
15.     public void write(int a, int b) {  
16.         synchronized(resourceB) { // May deadlock here  
17.             synchronized(resourceA) {  
18.                 resourceA.value = a;  
19.                 resourceB.value = b;  
20.             }  
21.         }  
22.     }  
23. }
```

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

the writer thread will have resourceB, and both will get stuck waiting for the other.

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

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

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

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

Thread Livelock

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

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

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

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

In this situation, the threads are not completely deadlocked, but they are

making no progress. This is an extremely tricky problem to detect! The timing has to be just right, so you might find your code runs perfectly fine 99 percent of the time and livelocks 1 percent of the time. Fortunately, Java makes livelock hard to do; with ReentrantLock (more on this in [Chapter 11](#) where we talk about concurrency) and careful ordering of how your threads attempt to access locks, you will be unlikely to encounter this problem.

Thread Starvation

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

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

Race Conditions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Now we have two instances of `Show`. So when thread one then uses its instance of `Show` to book a seat by calling `show.bookSeat()` on “1A”, it succeeds. Similarly, when thread two uses its instance of `Show` to book a seat, it too succeeds. We had two threads racing to get what should have been one shared resource, and because of timing, we end up with two instances of a resource and a failure in our program logic.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

You've seen an example where a race condition led to two threads both getting seat "1A". There are also race conditions in which neither thread gets seat "1A". It's far better for neither thread to end up with a seat than for two threads to end up with the same seat (while the venue operator might be unhappy

having an empty, unsold seat, it's worse to have customers fighting over seats!). However, by fixing the race conditions in this code, we've enabled the seats to be sold to the threads properly, so only one thread gets any given seat.

CERTIFICATION OBJECTIVE

Thread Interaction (OCP Objectives 10.2 and 10.3)

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

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

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

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

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

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

Think of a computer-controlled machine that cuts pieces of fabric into different shapes and an application that allows users to specify the shape to cut.

The current version of the application has one thread, which loops, first asking the user for instructions, and then directs the hardware to cut the requested shape:

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

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

A simple solution is to separate the processes into two different threads, one of them interacting with the user and another managing the hardware. The user thread sends the instructions to the hardware thread and then goes back to interacting with the user immediately. The hardware thread receives the instructions from the user thread and starts directing the machine immediately. Both threads use a common object to communicate, which holds the current design being processed.

The following pseudocode shows this design:

```

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

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

```

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

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

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

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

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

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

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

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

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

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

have to take care of the exception:

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

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

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

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

The resulting code would look like this:

```

class Operator extends Thread {
    public void run(){
        while(true){
            // Get shape from user
            synchronized(this){
                // Calculate new machine steps from shape
                notify();
            }
        }
    }
}

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

```

The machine thread, once started, will immediately go into the waiting state and will wait patiently until the operator sends the first notification. At that

point, it is the operator thread that owns the lock for the object, so the hardware thread gets stuck for a while. It's only after the operator thread abandons the synchronized block that the hardware thread can really start processing the machine steps.

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

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

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

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



When the `wait()` method is invoked on an object, the thread executing that code gives up its lock on the object immediately. However, when

notify() is called, that doesn't mean the thread gives up its lock at that moment. If the thread is still completing synchronized code, the lock is not released until the thread moves out of synchronized code. So just because notify() is called, this doesn't mean the lock becomes available at that moment.

Using `notifyAll()` When Many Threads May Be Waiting

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

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

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

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

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

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

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

Using `wait()` in a Loop

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

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

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

```
class Machine extends Thread {  
    List<MachineInstructions> jobs =  
        new ArrayList<MachineInstructions>();  
  
    public void addJob(MachineInstructions job) {  
        synchronized (jobs) {  
            jobs.add(job);  
            jobs.notify();  
        }  
    }  
}
```

```

        }
    }

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

```

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

has called `notify()` or `notifyAll()`. (At least, no code you know about has called these methods. Sometimes, the JVM may call `notify()` for reasons of its own, or code in some other class calls it for reasons you just don't know.) What this means is that when your thread wakes up from a `wait()`, you don't know for sure why it was awakened. By putting the `wait()` method in a `while` loop and rechecking the condition that represents what we were waiting for, we ensure that *whatever* the reason we woke up, we will re-enter the `wait()` if (and only if) the thing we were waiting for has not happened yet. In the `Machine` class, the thing we were waiting for is for the jobs list to not be empty. If it's empty, we `wait`, and if it's not, we don't.

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

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



The methods `wait()`, `notify()`, and `notifyAll()` are methods of only `java.lang.Object`, not of `java.lang.Thread` or `java.lang.Runnable`. Be

sure you know which methods are defined in Thread, which in object, and which in Runnable (just run(), so that's an easy one). Of the key methods in Thread, be sure you know which are static—sleep() and yield()—and which are not static—join() and start(). Table 10-2 lists the key methods you'll need to know for the exam, with the static methods shown in italics.

TABLE 10-2

Key Thread Methods

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

CERTIFICATION SUMMARY

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

We looked at five thread states: new, runnable, running, blocked/waiting/sleeping, and dead. You learned that when a thread is dead, it can never be restarted even if it's still a valid object on the heap. We saw that there is only one way a thread can transition to running, and that's from

Runnable. However, once running, a thread can become dead, go to sleep, wait for another thread to finish, block on an object's lock, wait for a notification, or return to runnable.

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



TWO-MINUTE DRILL

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

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

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

Transitioning Between Thread States (OCP Objective 10.1)

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

Sleep, Yield, and Join (OCP Objective 10.1)

- Sleeping is used to delay execution for a period of time, and no locks are released when a thread goes to sleep.
- A sleeping thread is guaranteed to sleep for at least the time specified in the argument to the `sleep()` method (unless it's interrupted), but there is no guarantee as to when the newly awakened thread will actually return to running.
- The `sleep()` method is a static method that sleeps the currently executing thread's state. One thread *cannot* tell another thread to sleep.

- ❑ The `setPriority()` method gives `Thread` objects a priority of between 1 (low) and 10 (high). Priorities are not guaranteed, and not all JVMs recognize ten distinct priority levels—some levels may be treated as effectively equal.
- ❑ If not explicitly set, a thread's priority will have the same priority as the thread that created it.
- ❑ The `yield()` method *may* cause a running thread to back out if there are runnable threads of the same priority. There is no guarantee that this will happen, and there is no guarantee that when the thread backs out there will be a *different* thread selected to run. A thread might yield and then immediately reenter the running state.
- ❑ The closest thing to a guarantee is that at any given time, when a thread is running, it will usually not have a lower priority than any thread in the runnable state. If a low-priority thread is running when a high-priority thread enters runnable, the JVM will usually preempt the running low-priority thread and put the high-priority thread in.
- ❑ When one thread calls the `join()` method of another thread, the currently running thread will wait until the thread it joins with has completed. Think of the `join()` method as saying, “Hey, thread, I want to join on to the end of you. Let me know when you’re done, so I can enter the runnable state.”

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

- ❑ `synchronized` methods prevent more than one thread from accessing an object's critical method code simultaneously.
- ❑ You can use the `synchronized` keyword as a method modifier or to start a synchronized block of code.
- ❑ To synchronize a block of code (in other words, a scope smaller than the whole method), you must specify an argument that is the object whose lock you want to synchronize on.
- ❑ While only one thread can be accessing synchronized code of a particular instance, multiple threads can still access the same object's unsynchronized code.
- ❑ When a thread goes to sleep, its locks will be unavailable to other

threads.

- static methods can be synchronized using the lock from the `java.lang.Class` instance representing that class.

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

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

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

- Deadlocking is when thread execution grinds to a halt because the code is waiting for locks to be removed from objects.
- Deadlocking can occur when a locked object attempts to access another locked object that is trying to access the first locked object. In other words, both threads are waiting for each other’s locks to be released; therefore, the locks will *never* be released!
- Deadlocking is bad. Don’t do it.
- Livelocking is when thread execution grinds to a halt because the threads are too busy to make any progress. The threads are still working but can’t get anywhere.
- Thread starvation is when a thread can’t get access to a resource it needs

so it starves. It's still alive, but barely.

- A race condition is when two threads race to get the same shared resource, and the result (often wrong) depends on which thread gets there first.
- Race conditions are bad. Don't allow them to happen. Use the `volatile` keyword to protect variables that should be atomic, and the `synchronized` keyword to make sure only one thread at a time can run code that manages a resource.

Q SELF TEST

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

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

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

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

- A. `public class MyRunnable extends Runnable{public void run(){}}`
- B. `public class MyRunnable extends Object{public void run(){}}`
- C. `public class MyRunnable implements Runnable{public void run(){}}`
- D. `public class MyRunnable implements Runnable{void run(){}}`
- E. `public class MyRunnable implements Runnable{public void start(){}}`

2. Given:

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

What is the result of this code?

- A. Compilation fails
- B. 1..2..3..
- C. 0..1..2..3..
- D. 0..1..2..
- E. An exception occurs at runtime

3. Given:

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

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

- A. Each `String` in the array `lines` will output, with a one-second pause between lines
 - B. Each `String` in the array `lines` will output, with no pause in between because this method is not executed in a `Thread`
 - C. Each `String` in the array `lines` will output, and there is no guarantee that there will be a pause because `currentThread()` may not retrieve this thread
 - D. This code will not compile
 - E. Each `String` in the `lines` array will print, with at least a one-second pause between lines
4. Assume you have a class that holds two `private` variables: `a` and `b`. Which of the following pairs can prevent concurrent access problems in that class? (Choose all that apply.)

- A.

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

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

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

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

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

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

5. Given:

```

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

```

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

- A. It fails to compile because the `IllegalMonitorStateException` of `wait()` is not dealt with in line 7
 - B. 1 2 3
 - C. 1 3
 - D. 1 2
 - E. At runtime, it throws an `IllegalMonitorStateException` when trying to wait
 - F. It will fail to compile because it has to be synchronized on the `this` object
- 6.** Assume the following method is properly synchronized and called from a thread A on an object B:

`wait(2000);`

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

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

- B. After the lock on B is released, or after two seconds
 - C. Two seconds after object B is notified
 - D. Two seconds after lock B is released
7. Which are true? (Choose all that apply.)
- A. The `notifyAll()` method must be called from a synchronized context
 - B. To call `wait()`, an object must own the lock on the thread
 - C. The `notify()` method is defined in class `java.lang.Thread`
 - D. When a thread is waiting as a result of `wait()`, it releases its lock
 - E. The `notify()` method causes a thread to immediately release its lock
 - F. The difference between `notify()` and `notifyAll()` is that `notifyAll()` notifies all waiting threads, regardless of the object they're waiting on
8. Given this scenario: This class is intended to allow users to write a series of messages so that each message is identified with a timestamp and the name of the thread that wrote the message:

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

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

- A. This class is already thread-safe
- B. Replacing `StringBuilder` with `StringBuffer` will make this class

thread-safe

- C. Synchronize the `log()` method only
- D. Synchronize the `getContents()` method only
- E. Synchronize both `log()` and `getContents()`
- F. This class cannot be made thread-safe

9. Given:

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

What is the result of this code?

- A. It prints x and exits
- B. It prints x and never exits
- C. It prints XY and exits almost immediately
- D. It prints XY with a 10-second delay between x and Y
- E. It prints XY with a 10,000-second delay between x and Y
- F. The code does not compile
- G. An exception is thrown at runtime

10. Given:

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

What is the result?

- A. foo
- B. MyThread foo
- C. MyThread bar
- D. foo bar
- E. foo bar baz
- F. bar foo

G. Compilation fails

H. An exception is thrown at runtime

11. Given:

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

```

        int option = (int) (Math.random() * 4);
        switch (option) {
            case 0: x.a(); break;
            case 1: x.b(); break;
            case 2: y.a(); break;
            case 3: y.b(); break;
        }
    };
Thread thread1 = new Thread(runnable);
Thread thread2 = new Thread(runnable);
thread1.start();
thread2.start();
}
}

```

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

- A. x.a() in thread1, and x.a() in thread2
- B. x.a() in thread1, and x.b() in thread2
- C. x.a() in thread1, and y.a() in thread2
- D. x.a() in thread1, and y.b() in thread2
- E. x.b() in thread1, and x.a() in thread2
- F. x.b() in thread1, and x.b() in thread2
- G. x.b() in thread1, and y.a() in thread2
- H. x.b() in thread1, and y.b() in thread2
- I. Compilation fails due to an error in declaring the Runnable

12. Given:

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

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

laurel.start();
hardy.start();
}
}
```

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

- A. A
- B. B
- C. c
- D. D
- E. E
- F. F
- G. The answer cannot be reliably determined
- H. The code does not compile

13. Given:

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

And given the following five fragments:

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

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

- A. All five will compile
- B. Only one might produce the output 4 4
- C. Only one might produce the output 4 2
- D. Exactly two might produce the output 4 4
- E. Exactly two might produce the output 4 2
- F. Exactly three might produce the output 4 4
- G. Exactly three might produce the output 4 2

14. Given:

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

Which are true? (Choose all that apply.)

- A. Compilation fails
- B. The output could be r1 r2 m1 m2
- C. The output could be m1 m2 r1 r2
- D. The output could be m1 r1 r2 m2
- E. The output could be m1 r1 m2 r2
- F. An exception is thrown at runtime

15. Given:

```
3. class Dudes {
4.     static long flag = 0;
5.     // insert code here
6.     if(flag == 0) flag = id;
7.     for(int x = 1; x < 3; x++) {
8.         if(flag == id) System.out.print("yo ");
9.         else System.out.print("dude ");
10.    }
11. }
12. }

13. public class DudesChat implements Runnable {
14.     static Dudes d;
15.     public static void main(String[] args) {
16.         new DudesChat().go();
17.     }
18.     void go() {
19.         d = new Dudes();
20.         new Thread(new DudesChat()).start();
21.         new Thread(new DudesChat()).start();
22.     }
23.     public void run() {
24.         d.chat(Thread.currentThread().getId());
25.     }
26. }
```

And given these two fragments:

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

When fragment I or fragment II is inserted at line 5, which are true?

(Choose all that apply.)

- A. An exception is thrown at runtime
- B. With fragment I, compilation fails
- C. With fragment II, compilation fails
- D. With fragment I, the output could be yo dude dude yo
- E. With fragment I, the output could be dude dude yo yo
- F. With fragment II, the output could be yo dude dude yo

16. Given:

```
3. class Chicks {  
4.     synchronized void yack(long id) {  
5.         for(int x = 1; x < 3; x++) {  
6.             System.out.print(id + " ");  
7.             Thread.yield();  
8.         }  
9.     }  
10. }  
11. public class ChicksYack implements Runnable {  
12.     Chicks c;  
13.     public static void main(String[] args) {  
14.         new ChicksYack().go();  
15.     }  
16.     void go() {  
17.         c = new Chicks();  
18.         new Thread(new ChicksYack()).start();  
19.         new Thread(new ChicksYack()).start();  
20.     }  
21.     public void run() {  
22.         c.yack(Thread.currentThread().getId());  
23.     }  
24. }
```

Which are true? (Choose all that apply.)

- A. Compilation fails
- B. The output could be 4 4 2 3
- C. The output could be 4 4 2 2

- D. The output could be 4 4 4 2
- E. The output could be 2 2 4 4
- F. An exception is thrown at runtime

17. Given:

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

And given these two fragments:

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

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

- A. Compilation fails
- B. With fragment I, an exception is thrown
- C. With fragment I, the output could be 4 2 4 2
- D. With fragment I, the output could be 4 4 2 3

- E. With fragment II, the output could be 2 4 2 4
18. You have two threads, t1 and t2, attempting to access a shared resource, and t2 is always descheduled when it tries to access that resource. What is this kind of problem called?
- A. A race condition
 - B. Deadlock
 - C. Livelock
 - D. Starvation
 - E. Synchronization
 - F. Multitasking

A SELF TEST ANSWERS

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

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

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

2. **D** is correct. The thread `MyThread` will start and loop three times (from 0 to 2).
- A** is incorrect because the `Thread` class implements the `Runnable` interface; therefore, in line 6, `Thread` can take an object of type `Thread` as an argument in the constructor (this is NOT recommended). **B** and **C** are incorrect because the variable `i` in the `for` loop starts with a value of 0 and ends with a value of 2. **E** is incorrect based on the above. (OCP Objective 10.1)
3. **D** is correct. The `sleep()` method must be enclosed in a `try/catch` block, or the method `printAll()` must declare it throws the

`InterruptedException`.

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

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

☒ **A** is incorrect because it is not synchronized; therefore, there is no guarantee that the values added by the `read()` method belong to the same pair. **C** and **D** are incorrect; only objects can be used to synchronize on. **E** is incorrect because it fails to compile—it is not possible to select other objects (even `this`) to synchronize on when declaring a method as synchronized. (OCP Objectives 10.2 and 10.3)
5. **D** is correct. 1 and 2 will be printed, but there will be no return from the `wait` call because no other thread will notify the main thread, so 3 will never be printed. It's frozen at line 7.

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

☒ **B** is incorrect because a waiting thread will not return to runnable when the lock is released unless a notification occurs. **C** is incorrect because the thread will become a candidate immediately after notification. **D** is also incorrect because a thread will not come out of a waiting pool just because a lock has been released. (OCP Objective 10.3)
7. **A** is correct because `notifyAll()` (and `wait()` and `notify()`) must be called from within a synchronized context. **D** is a correct statement.

B is incorrect because to call `wait()`, the thread must own the lock on the object that `wait()` is being invoked on, not the other way around. **C** is incorrect because `notify()` is defined in `java.lang.Object`. **E** is incorrect because `notify()` will not cause a thread to release its locks. The thread can only release its locks by exiting the synchronized code. **F** is incorrect because `notifyAll()` notifies all the threads waiting on a particular locked object, not all threads waiting on *any* object. (OCP Objectives 10.2 and 10.3)

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

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

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

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

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

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

- 11.** **A, F, and H** are correct. **A** is correct because when synchronized instance methods are called on the same *instance*, they block each other. **F** and **H** can't happen because synchronized static methods in the same

class block each other, regardless of which instance was used to call the methods. (An instance is not required to call static methods; only the class.)

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

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

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

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

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

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

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

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

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

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

A, B, C, D, and E are incorrect based on the above. (OCP Objectives 10.1 and 10.3)
17. **C** and **E** are correct. **E** should be obvious. **C** is correct because, even though `move()` is synchronized, it's being invoked on two different objects.

A, B, and D are incorrect based on the above. (OCP Objective 10.3)
18. **D** is correct. Starvation occurs when one or more threads cannot get access to a resource.

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

EXERCISE ANSWERS

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

The final code should look something like this:

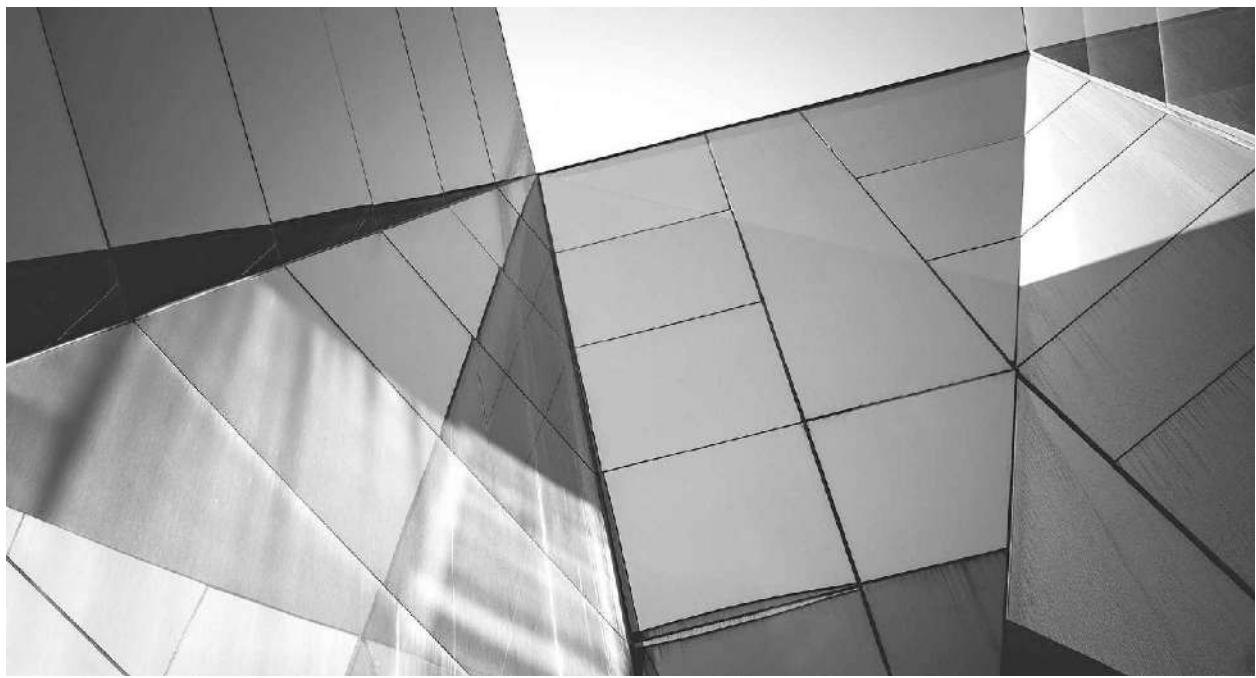
```
class TheCount extends Thread {  
    public void run() {  
        for(int i = 1;i<=100;++i) {  
            System.out.print(i + " ");  
            if(i % 10 == 0) System.out.println("Hahaha");  
            try { Thread.sleep(1000); }  
            catch(InterruptedException e) {}  
        }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

Exercise 10-2: Synchronizing a Block of Code

Your code might look something like this when completed:

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

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



11

Concurrency

CERTIFICATION OBJECTIVES

- Create Worker Threads Using Runnable, Callable and Use an ExecutorService to Concurrently Execute Tasks
 - Use Synchronized Keyword and java.util.concurrent.atomic Package to Control the Order of Thread Execution
 - Use java.util.concurrent Collections and Classes Including CyclicBarrier and CopyOnWriteArrayList
 - Use Parallel Fork/Join Framework
 - Use Parallel Streams Including Reduction, Decomposition, Merging Processes, Pipelines and Performance
 - Identify Potential Threading Problems among Deadlock, Starvation, Livelock, and Race Conditions
- ✓ Two-Minute Drill

Q&A Self Test

Concurrency with the `java.util.concurrent` Package

As you learned in the previous chapter on threads, the Java platform supports multithreaded programming. Supporting multithreaded programming is essential for any modern programming language because servers, desktop computers, laptops, and most mobile devices contain multiple CPUs. If you want your applications to take advantage of all of the processing power present in a modern system, you must create multithreaded applications.

Unfortunately, creating efficient and error-free multithreaded applications can be a challenge. The low-level threading constructs such as `Thread`, `Runnable`, `wait()`, `notify()`, and synchronized blocks are too primitive for many requirements and force developers to create their own high-level threading libraries. Custom threading libraries can be both error prone and time consuming to create.

The `java.util.concurrent` package provides high-level APIs that support many common concurrent programming use cases. When possible, you should use these high-level APIs in place of the traditional low-level threading constructs (`synchronized`, `wait`, `notify`). Some features (such as the locking API) provide functionality similar to what existed already, but with more flexibility at the cost of slightly awkward syntax. Using the `java.util.concurrent` classes requires a solid understanding of the traditional Java threading types (`Thread` and `Runnable`) and their use (`start`, `run`, `synchronized`, `wait`, `notify`, `join`, `sleep`, etc.). If you are not comfortable with Java threads, you should return to the previous chapter before continuing with these high-level concurrency APIs.

CERTIFICATION OBJECTIVE

Apply Atomic Variables and Locks (OCP Objective 10.3)

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

The `java.util.concurrent.atomic` and `java.util.concurrent.locks` packages solve two different problems. They are grouped into a single exam objective simply because they are the only two packages below `java.util.concurrent` and both have a small number of classes and interfaces to learn. The `java.util.concurrent.atomic` package enables multithreaded applications to safely access individual variables without locking, whereas the `java.util.concurrent.locks` package provides a locking framework that can be used to create locking behaviors that are the same or superior to those of Java's `synchronized` keyword.

Atomic Variables

Imagine a multiplayer video game that contains monsters that must be destroyed. The players of the game (threads) are vanquishing monsters, while at the same time a monster-spawning thread is repopulating the world to ensure players always have a new challenge to face. To keep the level of difficulty consistent, you would need to keep track of the monster count and ensure that the monster population stays the same (a hero's work is never done). Both the player threads and the monster-spawning thread must access and modify the shared monster count variable. If the monster count somehow became incorrect, your players may find themselves with more adversaries than they could handle.

The following example shows how even the seemingly simplest of code can lead to undefined results. Here you have a class that increments and reports the current value of an integer variable:

```
public class Counter {  
    private int count;  
    public void increment() {  
        count++; // it's a trap!  
        // a single "line" is not atomic  
    }  
    public int getValue() {  
        return count;  
    }  
}
```

A Thread that will increment the counter 10,000 times:

```
public class IncrementerThread extends Thread {  
    private Counter counter;  
    // all instances are passed the same counter  
    public IncrementerThread(Counter counter) {  
        this.counter = counter;  
    }  
    public void run() {  
        // "i" is local and therefore thread-safe  
        for(int i = 0; i < 10000; i++) {  
            counter.increment();  
        }  
    }  
}
```

Here is the code from within this application's main method:

```
Counter counter = new Counter();          // the shared object  
IncrementerThread it1 = new IncrementerThread(counter);  
IncrementerThread it2 = new IncrementerThread(counter);  
it1.start(); // thread 1 increments the count by 10000  
it2.start(); // thread 2 increments the count by 10000  
it1.join(); // wait for thread 1 to finish  
it2.join(); // wait for thread 2 to finish  
System.out.println(counter.getValue()); // rarely 20000  
                                         // lowest 11972
```

The trap in this example is that `count++` looks like a single action when, in fact, it is not. When incrementing a field like this, what *probably* happens is the

following sequence:

1. The value stored in count is copied to a temporary variable.
2. The temporary variable is incremented.
3. The value of the temporary variable is copied back to the count field.

We say “probably” in this example because while the Java compiler will translate the `count++` statement into multiple Java bytecode instructions, you really have no control over what native instructions are executed. The JIT (Just In Time compiler)-based nature of most Java runtime environments means you don’t know when or if the `count++` statement will be translated to native CPU instructions and whether it ends up as a single instruction or several. You should always act as if a single line of Java code takes multiple steps to complete. Getting an incorrect result also depends on many other factors, such as the type of CPU you have. Do both threads in the example run concurrently or in sequence? A large loop count was used in order to make the threads run longer and be more likely to execute concurrently.

While you could make this code thread-safe with synchronized blocks, the act of obtaining and releasing a lock flag would probably be more time consuming than the work being performed. This is where the classes in the `java.util.concurrent.atomic` package can benefit you. They provide variables whose values can be modified atomically. An atomic operation is one that, for all intents and purposes, appears to happen all at once. The `java.util.concurrent.atomic` package provides several classes for different data types, such as `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, and `AtomicReference`, to name a few.

Here is a thread-safe replacement for the Counter class from the previous example:

```
public class Counter {  
    private AtomicInteger count = new AtomicInteger();  
    public void increment() {  
        count.getAndIncrement(); // atomic operation  
    }  
    public int getValue() {  
        return count.intValue();  
    }  
}
```

In reality, even a method such as `getAndIncrement()` still takes several steps to execute. The reason this implementation is now thread-safe is something called CAS. CAS stands for Compare And Swap. Most modern CPUs have a set of CAS instructions. Following is a basic outline of what is happening now:

1. The value stored in `count` is copied to a temporary variable.
2. The temporary variable is incremented.
3. Compare the value currently in `count` with the original value. If it is unchanged, then swap the old value for the new value.

Step 3 happens atomically. If step 3 finds that some other thread has already modified the value of `count`, then repeat steps 1–3 until we increment the field without interference.

The central method in a class like `AtomicInteger` is the boolean `compareAndSet(int expect, int update)` method, which provides the CAS behavior. Other atomic methods delegate to the `compareAndSet` method. The `getAndIncrement` method implementation is simply:

```
public final int getAndIncrement() {  
    for (;;) {  
        int current = get();  
        int next = current + 1;  
        if (compareAndSet(current, next))  
            return current;  
    }  
}
```

Locks

The `java.util.concurrent.locks` package is about creating (not surprisingly) locks. Why would you want to use locks when so much of `java.util.concurrent` seems geared toward avoiding overt locking? You use `java.util.concurrent.locks` classes and traditional monitor locking (the `synchronized` keyword) for roughly the same purpose: creating segments of code that require exclusive execution (one thread at a time).

Why would you create code that limits the number of threads that can execute it? While atomic variables work well for making single variables thread-safe, imagine if you have two or more variables that are related. A video game character might have a number of gold pieces that can be carried in his backpack and a number of gold pieces he keeps in an in-game bank vault. Transferring gold into the bank is as simple as subtracting gold from the backpack and adding it to the vault. If we have 10 gold pieces in our backpack and 90 in the vault, we have a total of 100 pieces that belong to our character. If we want to transfer all 10 pieces to the vault, we can first add 10 to the vault count and then subtract 10 from the backpack, or first subtract 10 from the backpack and then add 10 to the vault. If another thread were to try to assess our character's wealth during the middle of our transfer, it might see 90 pieces or 110 pieces, depending on the order of our operations, neither being the correct count of 100 pieces.

This other thread that is attempting to read the character's total wealth might do all sorts of things, such as increase the likelihood of your character being robbed or a variety of other actions to control the in-game economics. All game threads must correctly gauge a character's wealth even if there is a transfer in progress.

The solution to our balance inquiry transfer problem is to use locking. Create a single method to get a character's wealth and another to perform gold transfers. You should never be able to check a character's total wealth while a gold transfer is in progress. Having a single method to get a character's total wealth is also important because you don't want a thread to read the backpack's gold count before a transfer and then the vault's gold count after a transfer. That would lead to the same incorrect total as trying to calculate the total during a transfer.

Much of the functionality provided by the classes and interfaces of the `java.util.concurrent.locks` package duplicates that of traditional synchronized locking. In fact, the hypothetical gold transfer outlined earlier could be solved with either the `synchronized` keyword or classes in the `java.util.concurrent.locks` package. In Java 5, when `java.util.concurrent` was first introduced, the new locking classes performed better than the `synchronized` keyword, but there is no longer a vast difference in performance. So why would you use these newer locking classes? The `java.util.concurrent.locks` package provides

- The ability to duplicate traditional synchronized blocks.
- Nonblock scoped locking—obtain a lock in one method and release it in another (this can be dangerous, though).
- Multiple `wait/notify/notifyAll` pools per lock—threads can select which pool (condition) they wait on.
- The ability to attempt to acquire a lock and take an alternative action if locking fails.
- An implementation of a multiple-reader, single-writer lock.

ReentrantLock

The `java.util.concurrent.locks.Lock` interface provides the outline of locking provided by the `java.util.concurrent.locks` package. Like any interface, the `Lock` interface requires an implementation to be of any real use. The `java.util.concurrent.locks.ReentrantLock` class provides that implementation. To demonstrate the use of `Lock`, we will first duplicate the functionality of a basic traditional synchronized block.

```
Object obj = new Object();
synchronized(obj) {    // traditional locking, blocks until acquired
                      // work
}
                      // releases lock automatically
```

Here is an equivalent piece of code using the `java.util.concurrent.locks` package. Notice how `ReentrantLock` can be stored in a `Lock` reference because it implements the `Lock` interface. This example blocks on attempting to acquire a lock, just like traditional synchronization.

```
Lock lock = new ReentrantLock();
lock.lock();           // blocks until acquired
try {
    // do work here
} finally {            // to ensure we unlock
    lock.unlock();     // must manually release
}
```

It is recommended that you always follow the `lock()` method with a `try-finally` block, which releases the lock. The previous example doesn't really provide a compelling reason for you to choose to use a `Lock` instance instead of traditional synchronization. One of the very powerful features is the ability to attempt (and fail) to acquire a lock. With traditional synchronization, once you hit a `synchronized` block, your thread either immediately acquires the lock or blocks until it can.

```
Lock lock = new ReentrantLock();
boolean locked = lock.tryLock(); // try without waiting
if (locked) {
    try {
        // work
    } finally {                                // to ensure we unlock
        lock.unlock();
    }
}
```

The ability to quickly fail to acquire the lock turns out to be powerful. You can process a different resource (lock) and come back to the failed lock later instead of just waiting for a lock to be released and thereby making more efficient use of system resources. There is also a variation of the `tryLock` method that allows you to specify an amount of time you are willing to wait to acquire the lock:

```
Lock lock = new ReentrantLock();
try {
    boolean locked = lock.tryLock(3, TimeUnit.SECONDS);
    if (locked) {
        try {
            // work
        } finally {          // to ensure we unlock
            lock.unlock();
        }
    }
} catch (InterruptedException ex) {
    // handle
}
```

Another benefit of the `tryLock` method is deadlock avoidance. With traditional synchronization, you must acquire locks in the same order across all threads. For example, if you have two objects to lock against:

```
Object o1 = new Object();
Object o2 = new Object();
```

and you synchronize using the internal lock flags of both objects:

```
synchronized(o1) {
    // thread A could pause here
    synchronized(o2) {
        // work
    }
}
```

you should never acquire the locks in the opposite order because it could lead to deadlock. Although thread A has only the o1 lock, thread B acquires the o2 lock. You are now at an impasse because neither thread can obtain the second lock it needs to continue.

```
synchronized(o2) {  
    // thread B gets stuck here  
    synchronized(o1) {  
        // work  
    }  
}
```

Looking at a similar example using a ReentrantLock, start by creating two locks:

```
Lock l1 = new ReentrantLock();  
Lock l2 = new ReentrantLock();
```

Next, you acquire both locks in thread A:

```
boolean aq1 = l1.tryLock();  
boolean aq2 = l2.tryLock();  
try{  
    if (aq1 && aq2) {  
        // work  
    }  
} finally {  
    if (aq2) l2.unlock(); // don't unlock if not locked  
    if (aq1) l1.unlock();  
}
```

Notice the example is careful to always unlock any acquired lock, but ONLY the lock(s) that were acquired. A `ReentrantLock` has an internal counter that keeps track of the number of times it has been locked/unlocked, and it is an error to unlock without a corresponding successful lock operation. If a thread attempts to release a lock that it does not own, an `IllegalMonitorStateException` will be thrown.

Now in thread B, the locks are obtained in the reverse order in which thread A obtained them. With traditional locking, using synchronized code blocks and attempting to obtain locks in the reverse order could lead to deadlock.

```
boolean aq2 = l2.tryLock();
boolean aq1 = l1.tryLock();
try{
    if (aq1 && aq2) {
        // work

    }
} finally {
    if (aq1) l1.unlock();
    if (aq2) l2.unlock();
}
```

Now, even if thread A was only in possession of the `l1` lock, there is no possibility that thread B could block because we use the nonblocking `tryLock` method. Using this technique, you can avoid deadlocking scenarios, but you must deal with the possibility that both locks could not be acquired. Using a simple loop, you can repeatedly attempt to obtain both locks until successful (Note: This approach is CPU intensive; we'll look at a better solution later):

```

loop2:
while (true) {
    boolean aq2 = l2.tryLock();
    boolean aq1 = l1.tryLock();
    try {
        if (aq1 && aq2) {
            // work
            break loop2;
        }
    } finally {
        if (aq2) l2.unlock();
        if (aq1) l1.unlock();
    }
}

```



It is remotely possible that this example could lead to livelock. Imagine if thread A always acquires lock1 at the same time that thread B acquires lock2. Each thread's attempt to acquire the second lock would always fail, and you'd end up repeating forever, or at least until you were lucky enough to have one thread fall behind the other. You can avoid livelock in this scenario by introducing a short random delay with `Thread.sleep(int)` any time you fail to acquire both locks.

Condition

A Condition provides the equivalent of the traditional wait, notify, and notifyAll methods. The traditional wait and notify methods allow developers to implement an await/signal pattern. You use an await/signal pattern when you

would use locking, but with the added stipulation of trying to avoid spinning (endless checking if it is okay to do something). Imagine a video game character who wants to buy something from a store, but the store is out of stock at the moment. The character’s thread could repeatedly lock the store object and check for the desired item, but that would lead to unneeded system utilization. Instead, the character’s thread can say, “I’m taking a nap, wake me up when new stock arrives.”

The `java.util.concurrent.locks.Condition` interface is a replacement for the `wait` and `notify` methods. A three-part code example shows you how to use a condition. Part one shows that a `Condition` is created from a `Lock` object:

```
Lock lock = new ReentrantLock();
Condition blockingPoolA = lock.newCondition();
```

When your thread reaches a point where it must delay until another thread performs an activity, you “await” the completion of that other activity. Before calling `await`, you must have locked the `Lock` used to produce the `Condition`. It is possible that the awaiting thread may be interrupted, and you must handle the possible `InterruptedException`. When you call the `await` method, the `Lock` associated with the `Condition` is released. Before the `await` method returns, the lock will be reacquired. In order to use a `Condition`, a thread must first acquire a `Lock`. Part two of the three-part `Condition` example shows how a `Condition` is used to pause or wait for some event:

```
lock.lock();
try {
    blockingPoolA.await(); // "wait" here
                           // lock will be reacquired
    // work
} catch (InterruptedException ex) {
    // interrupted during await()
} finally {
    // to ensure we unlock
    lock.unlock();          // must manually release
}
```

In another thread, you perform the activity that the first thread was waiting on and then signal that first thread to resume (return from the `await` method). Part three of the `Condition` example is run in a different thread than part two. This part causes the thread waiting in the second piece to wake up:

```
lock.lock();
try {
    // work
    blockingPoolA.signalAll(); // wake all awaiting
                               // threads
} finally {
    lock.unlock();           // now an awoken thread can run
}
```

The `signalAll()` method causes all threads awaiting on the same `Condition` to wake up. You can also use the `signal()` method to wake up a single awaiting thread. Remember that “waking up” is not the same thing as proceeding. Each awoken thread will have to reacquire the `Lock` before continuing.

One advantage of a `Condition` over the traditional `wait/notify` operations is that multiple `Conditions` can exist for each `Lock`. A `Condition` is effectively a waiting/blocking pool for threads.

```
Lock lock = new ReentrantLock();
Condition blockingPoolA = lock.newCondition();
Condition blockingPoolB = lock.newCondition();
```

By having multiple conditions, you are effectively categorizing the threads waiting on a lock and can, therefore, wake up a subset of the waiting threads.

Conditions can also be used when you can't use a `BlockingQueue` to coordinate the activities of two or more threads.

ReentrantReadWriteLock

Imagine a video game that was storing a collection of high scores using a non-thread-safe collection. With a non-thread-safe collection, it is important that if a

thread is attempting to modify the collection, it must have exclusive access to the collection. To allow multiple threads to concurrently read the high score list or allow a single thread to add a new score, you could use a `ReadWriteLock`.

A `ReentrantReadWriteLock` is not actually a `Lock`; it implements the `ReadWriteLock` interface. What a `ReentrantReadWriteLock` does is produce two specialized `Lock` instances, one to a read lock and the other to a write lock.

```
ReentrantReadWriteLock rwl =  
    new ReentrantReadWriteLock();  
Lock readLock = rwl.readLock();  
Lock writeLock = rwl.writeLock();
```

These two locks are a matched set—one cannot be held at the same time as the other (by different threads). What makes these locks unique is that multiple threads can hold the read lock at the same time, but only one thread can hold the write lock at a time.

This example shows how a non-thread-safe collection (an `ArrayList`) can be made thread-safe, allowing concurrent reads but exclusive access by a writing thread:

```
public class MaxValueCollection {  
    private List<Integer> integers = new ArrayList<>();  
    private ReentrantReadWriteLock rwl =  
        new ReentrantReadWriteLock();
```

```

public void add(Integer i) {
    rwl.writeLock().lock(); // one at a time
    try {
        integers.add(i);
    } finally {
        rwl.writeLock().unlock();
    }
}

public int findMax() {
    rwl.readLock().lock(); // many at once
    try {
        return Collections.max(integers);
    } finally {
        rwl.readLock().unlock();
    }
}
}

```

Instead of wrapping a collection with `Lock` objects to ensure thread safety, you can use one of the thread-safe collections you'll learn about in the next section.

CERTIFICATION OBJECTIVE

Use `java.util.concurrent` Collections (OCP Objective 10.4)

10.4 Use `java.util.concurrent` collections and classes including `CyclicBarrier`

and CopyOnWriteArrayList.

Imagine an online video game with a list of the top 20 scores in the last 30 days. You could model the high score list using a `java.util.ArrayList`. As scores expire, they are removed from the list, and as new scores displace existing scores, remove and insert operations are performed. At the end of every game, the list of high scores is displayed. If the game is popular, then a lot of people (threads) will be reading the list at the same time. Occasionally, the list will be modified—sometimes by multiple threads—probably at the same time that it is being read by a large number of threads.

A traditional `java.util.List` implementation such as `java.util.ArrayList` is not thread-safe. Concurrent threads can safely read from an `ArrayList` and possibly even modify the elements stored in the list, but if any thread modifies the structure of the list (add or remove operation), then unpredictable behavior can occur.

Look at the `ArrayListRunnable` class in the following example. What would happen if there were a single instance of this class being executed by several threads? You might encounter several problems, including `ArrayIndexOutOfBoundsException`, duplicate values, skipped values, and null values. Not all threading problems manifest immediately. To observe the bad behavior, you might have to execute the faulty code multiple times or under different system loads. It is important that you are able to recognize the difference between thread-safe and non-thread-safe code yourself, because the compiler will not detect thread-unsafe code.

```
public class ArrayListRunnable implements Runnable {
    // shared by all threads
    private List<Integer> list = new ArrayList<>();

    public ArrayListRunnable() {
        // add some elements
        for (int i = 0; i < 100000; i++) {
            list.add(i);
        }
    }

    // might run concurrently, you cannot be sure
    // to be safe you must assume it does
    public void run() {
        String tName = Thread.currentThread().getName();
        while (!list.isEmpty()) {
            System.out.println(tName + " removed " + list.remove(0));
        }
    }

    public static void main(String[] args) {
        ArrayListRunnable alr = new ArrayListRunnable();
        Thread t1 = new Thread(alr);
        Thread t2 = new Thread(alr); // shared Runnable
        t1.start();
        t2.start();
    }
}
```

To make a collection thread-safe, you could surround all the code that accessed the collection in synchronized blocks or use a method such as `Collections.synchronizedList(new ArrayList())`. Using synchronization to safeguard a collection creates a performance bottleneck and reduces the liveness of your application. The `java.util.concurrent` package provides several types of collections that are thread-safe but do not use coarse-grained synchronization. When a collection will be concurrently accessed in an application you are developing, you should always consider using the collections outlined in the following sections.



Problems in multithreaded applications may not always manifest—a lot depends on the underlying operating system and how other applications affect the thread scheduling of a problematic application. On the exam, you might be asked about the “probable” or “most likely” outcome. Unless you are asked to identify every possible outcome of a code sample, don’t get hung up on unlikely results. For example, if a code sample uses `Thread.sleep(1000)` and nothing indicates that the thread would be interrupted while it was sleeping, it would be safe to assume that the thread would resume execution around one second after the call to sleep.

Copy-on-Write Collections

The copy-on-write collections from the `java.util.concurrent` package implement one of several mechanisms to make a collection thread-safe. By using the copy-on-write collections, you eliminate the need to implement synchronization or locking when manipulating a collection using multiple threads.

The `CopyOnWriteArrayList` is a `List` implementation that can be used concurrently without using traditional synchronization semantics. As its name implies, a `CopyOnWriteArrayList` will never modify its internal array of data. Any mutating operations on the `List` (`add`, `set`, `remove`, etc.) will cause a new modified copy of the array to be created, which will replace the original read-only array. The read-only nature of the underlying array in a

`CopyOnWriteArrayList` allows it to be safely shared with multiple threads.

Remember that read-only (immutable) objects are always thread-safe.

The essential thing to remember with a copy-on-write collection is that a thread that is looping through the elements in a collection must keep a reference to the same unchanging elements throughout the duration of the loop; this is achieved with the use of an `Iterator`. You want to keep using the old, unchanging collection that you began a loop with. When you use `list.iterator()`, the returned `Iterator` will always reference the collection of elements as it was when `list.iterator()` was called, even if another thread modifies the collection. Any mutating methods called on a copy-on-write-based `Iterator` or `ListIterator` (such as `add`, `set`, or `remove`) will throw an `UnsupportedOperationException`.



A `for-each` loop uses an `Iterator` when executing, so it is safe to use with a copy-on-write collection, unlike a traditional `for` loop.

```
for(Object o : collection) {} // use this  
for(int i = 0; i < collection.size(); i++) {} // not this
```

The `java.util.concurrent` package provides two copy-on-write-based collections: `CopyOnWriteArrayList` and `CopyOnWriteArraySet`. Use the copy-on-write collections when your data sets remain relatively small and the number of read operations and traversals greatly outnumber modifications to the collections. Modifications to the collections (not the elements within) are expensive because the entire internal array must be duplicated for each modification.



A thread-safe collection does not make the elements stored within the collection thread-safe. Just because a collection that contains elements is threadsafe does not mean the elements themselves can be safely modified

by multiple threads. You might have to use atomic variables, locks, synchronized code blocks, or immutable (read-only) objects to make the objects referenced by a collection thread-safe.

Concurrent Collections

The `java.util.concurrent` package also contains several concurrent collections that can be concurrently read and modified by multiple threads, but without the copy-on-write behavior seen in the copy-on-write collections. The concurrent collections include

- `ConcurrentHashMap`
- `ConcurrentLinkedDeque`
- `ConcurrentLinkedQueue`
- `ConcurrentSkipListMap`
- `ConcurrentSkipListSet`

Be aware that an `Iterator` for a concurrent collection is weakly consistent; it can return elements from the point in time the `Iterator` was created **or later**. This means that while you are looping through a concurrent collection, you might observe elements that are being inserted by other threads. In addition, you may observe only some of the elements that another thread is inserting with methods such as `addAll` when concurrently reading from the collection. Similarly, the `size` method may produce inaccurate results. Imagine attempting to count the number of people in a checkout line at a grocery store. While you are counting the people in line, some people may join the line and others may leave. Your count might end up close but not exact by the time you reach the end. This is the type of behavior you might see with a weakly consistent collection. The benefit to this type of behavior is that it is permissible for multiple threads to concurrently read and write a collection without having to create multiple internal copies of the collection, as is the case in a copy-on-write collection. If your application cannot deal with these inconsistencies, you might have to use a copy-on-write collection.

The `ConcurrentHashMap` and `ConcurrentSkipListMap` classes implement the `ConcurrentMap` interface. A `ConcurrentMap` enhances a `Map` by adding the atomic `putIfAbsent`, `remove`, and `replace` methods. For example, the `putIfAbsent` method is equivalent to performing the following code as an

atomic operation:

```
if (!map.containsKey(key))  
    return map.put(key, value);  
else  
    return map.get(key);
```

ConcurrentSkipListMap and ConcurrentSkipListSet are sorted. ConcurrentSkipListMap keys and ConcurrentSkipListSet elements require the use of the Comparable or Comparator interfaces to enable ordering.

Blocking Queues

The copy-on-write and the concurrent collections are centered on the idea of multiple threads sharing data. Sometimes, instead of shared data (objects), you need to transfer data between two threads. A `BlockingQueue` is a type of shared collection that is used to exchange data between two or more threads while causing one or more of the threads to wait until the point in time when the data can be exchanged. One use case of a `BlockingQueue` is called the producer-consumer problem. In a producer-consumer scenario, one thread produces data, then adds it to a queue, and another thread must consume the data from the queue. A queue provides the means for the producer and the consumer to exchange objects. The `java.util.concurrent` package provides several `BlockingQueue` implementations. They include

- `ArrayBlockingQueue`
- `LinkedBlockingDeque`
- `LinkedBlockingQueue`
- `PriorityBlockingQueue`
- `DelayQueue`
- `LinkedTransferQueue`
- `SynchronousQueue`

General Behavior

A blocking collection, depending on the method being called, may cause a thread

to block until another thread calls a corresponding method on the collection. For example, if you attempt to remove an element by calling `take()` on any `BlockingQueue` that is empty, the operation will block until another thread inserts an element. Don't call a blocking operation in a thread unless it is safe for that thread to block. The commonly used methods in a `BlockingQueue` are described in the following table.

Method	General Purpose	Unique Behavior
<code>add(E e)</code>	Insert an object.	Returns <code>true</code> if object added, <code>false</code> if duplicate objects are not allowed. Throws an <code>IllegalStateException</code> if the queue is bounded and full.
<code>offer(E e)</code>	Insert an object.	Returns <code>true</code> if object added, <code>false</code> if the queue is bounded and full.
<code>put(E e)</code>	Insert an object.	Returns <code>void</code> . If needed, will block until space in the queue becomes available.
<code>offer(E e, long timeout, TimeUnit unit)</code>	Insert an object.	Returns <code>false</code> if the object was not able to be inserted before the time indicated by the second and third parameters.
<code>remove(Object o)</code>	Remove an object.	Returns <code>true</code> if an equal object was found in the queue and removed; otherwise, returns <code>false</code> .

Method	General Purpose	Unique Behavior
<code>poll(long timeout, TimeUnit unit)</code>	Remove an object.	Removes the first object in the queue (the head) and returns it. If the timeout expires before an object can be removed because the queue is empty, a null will be returned.
<code>take()</code>	Remove an object.	Removes the first object in the queue (the head) and returns it, blocking if needed until an object becomes available.
<code>poll()</code>	Remove an object.	Removes the first object in the queue (the head) and returns it or returns null if the queue is empty.
<code>element()</code>	Retrieve an object.	Gets the head of the queue without removing it. Throws a <code>NoSuchElementException</code> if the queue is empty.
<code>peek()</code>	Retrieve an object.	Gets the head of the queue without removing it. Returns a null if the queue is empty.

Bounded Queues

`ArrayBlockingQueue`, `LinkedBlockingDeque`, and `LinkedBlockingQueue` support a bounded capacity and will block on `put(e)` and similar operations if the collection is full. `LinkedBlockingQueue` is optionally bounded, depending on the constructor you use.

```
BlockingQueue<Integer> bq = new ArrayBlockingQueue<>(1);
try {
    bq.put(42);
    bq.put(43); // blocks until previous value is removed
} catch (InterruptedException ex) {
    // log and handle
}
```

Special-Purpose Queues

A `SynchronousQueue` is a special type of bounded blocking queue; it has a capacity of zero. Having a zero capacity, the first thread to attempt either an insert or remove operation on a `SynchronousQueue` will block until another thread performs the opposite operation. You use a `SynchronousQueue` when you need threads to meet up and exchange an object.

A `DelayQueue` is useful when you have objects that should not be consumed until a specific time. The elements added to a `DelayQueue` will implement the `java.util.concurrent.Delayed` interface, which defines a single method: `public long getDelay(TimeUnit unit)`. The elements of a `DelayQueue` can only be taken once their delay has expired.

The `LinkedTransferQueue`

A `LinkedTransferQueue` (added in Java 7) is a superset of `ConcurrentLinkedQueue`, `SynchronousQueue`, and `LinkedBlockingQueue`. It can function as a concurrent Queue implementation similar to `ConcurrentLinkedQueue`. It also supports unbounded blocking (consumption blocking) similar to `LinkedBlockingQueue` via the `take()` method. Like a `SynchronousQueue`, a `LinkedTransferQueue` can be used to make two threads rendezvous to exchange an object. Unlike a `SynchronousQueue`, a `LinkedTransferQueue` has internal capacity, so the `transfer(E)` method is used to block until the inserted object (and any previously inserted objects) is consumed by another thread.

In other words, a `LinkedTransferQueue` might do almost everything you need from a Queue.

Because a `LinkedTransferQueue` implements the `BlockingQueue`,

`TransferQueue`, and `Queue` interfaces, it can be used to showcase all the different methods that can be used to add and remove elements using the various types of queues. Creating a `LinkedTransferQueue` is easy. Because `LinkedTransferQueue` is not bound by size, a limit to the number of elements CANNOT be supplied to its constructor.

```
TransferQueue<Integer> tq =  
    new LinkedTransferQueue<>(); // not bounded
```

There are many methods to add a single element to a `LinkedTransferQueue`. Note that any method that blocks or waits for any period may throw an `InterruptedException`.

```
boolean b1 = tq.add(1);           // returns true if added or throws  
                                // IllegalStateException if full  
tq.put(2);                     // blocks if bounded and full  
boolean b3 = tq.offer(3);        // returns true if added or false  
                                // if bounded and full  
                                // recommended over add  
boolean b4 =  
    tq.offer(4, 10, MILLISECONDS); // returns true if added  
                                // within the given time  
                                // false if bound and full  
tq.transfer(5);                // blocks until this element is consumed
```

```
boolean b6 = tq.tryTransfer(6);           // returns true if consumed
                                              // by an awaiting thread or
                                              // returns false without
                                              // adding if there was no
                                              // awaiting consumer

boolean b7 =
tq.tryTransfer(7, 10, MILLISECONDS); // will wait the
                                              // given time for
                                              // a consumer
```

Shown next are the various methods to access a single value in a `LinkedTransferQueue`. Again, any method that blocks or waits for any period may throw an `InterruptedException`.

```

Integer i1 = tq.element();      // gets without removing
                                // throws NoSuchElementException
                                // if empty
Integer i2 = tq.peek();        // gets without removing
                                // returns null if empty
Integer i3 = tq.poll();        // removes the head of the queue
                                // returns null if empty
Integer i4 =
    tq.poll(10, MILLISECONDS); // removes the head of the
                                // queue, waits up to the time
                                // specified before returning
                                // null if empty
Integer i5 = tq.remove();      // removes the head of the queue
                                // throws NoSuchElementException
                                // if empty
Integer i6 = tq.take();        // removes the head of the queue
                                // blocks until an element is ready

```



Use a `LinkedTransferQueue` (added in Java 7) instead of another comparable queue type. The other `java.util.concurrent` queues (introduced in Java 5) are less efficient than `LinkedTransferQueue`.

Controlling Threads with CyclicBarrier

Whereas blocking queues force threads to wait based on capacity or until a method is called on the queue, a `CyclicBarrier` can force threads to wait at a

specific point in the execution until all threads reach that point before continuing. You can think of that point in the execution as a *barrier*: no thread can proceed beyond that point until all other threads have also reached it.

Imagine you have two threads that are processing data in arrays and you want to copy the processed data from the arrays to a final `ArrayList` that contains data from both threads. You can't have both threads accessing the final `ArrayList` because `ArrayList` is not thread-safe. It is better to wait until both threads are done processing the arrays and only then copy the data to the final `ArrayList` using one thread.

This is an example of the type of problem where `CyclicBarrier` can be useful. When you can break up a problem into smaller pieces that can be processed concurrently and then combine the data into a result at certain key points in the processing, using a `CyclicBarrier` ensures the threads wait for each other at those key points in the execution.

Let's take a look at some code to see how to use `CyclicBarrier` to force two threads to wait for each other. In this example, we'll use the optional `Runnable` that `CyclicBarrier` will take and run, once both threads reach the barrier point. This `Runnable` is run only one time, by the last thread to reach the barrier and before any of the threads can continue, ensuring the code in the `Runnable` is thread-safe. Each thread processes a small array and then ends, but imagine how this idea might be applied to threads processing large arrays a chunk at a time and continuing after a barrier is reached. In fact, that's exactly why `CyclicBarrier` is named "cyclic": the `CyclicBarrier` can be reused after the threads are released to continue running.

```
public class CB {
    List<String> result = new ArrayList<>();
    static String[] dogs1 = {"boi", "clover", "charis"};
    static String[] dogs2 = {"aiko", "zooey", "biscuit"};
    final CyclicBarrier barrier;           // The barrier for the threads
    class ProcessDogs implements Runnable { // Each thread will process
        String dogs[];                  // an array of dogs
        ProcessDogs(String[] d) { dogs = d; }
    }
    // #4
    public void run() {                  // Convert first chars into
                                         // uppercase
        for (int i = 0; i < dogs.length; i++) {
            String dogName = dogs[i];
            String newDogName = dogName.substring(0, 1).toUpperCase()
                + dogName.substring(1);
            dogs[i] = newDogName;
        }
    try {
    // #5
        barrier.await();           // Wait at the barrier
    } catch(InterruptedException | BrokenBarrierException e) {
        // The other thread must have been interrupted
        e.printStackTrace();
    }
}
```

```
// #7
        System.out.println(Thread.currentThread().getName() + " is done!");
    }
}
public CB() {
// #1 ----- the 2nd argument code runs later
    barrier = new CyclicBarrier(2, () -> {
// 2 threads, 1 Runnable
// #6
                                            // Copy results to list
        for (int i = 0; i < dogs1.length; i++) {
            result.add(dogs1[i]);           // add dogs from array 1
        }
        for (int i = 0; i < dogs2.length; i++) {
            result.add(dogs2[i]);           // add dogs from array 2
        }
                                            // print the thread name and
                                            // result
        System.out.println(Thread.currentThread().getName() +
                           " Result: " + result);
    });
// #2
    Thread t1 = new Thread(new ProcessDogs(dogs1));
    Thread t2 = new Thread(new ProcessDogs(dogs2));
// #3
    t1.start();
    t2.start();
    System.out.println("Main Thread is done");
}
public static void main(String[] args) {
    CB cb = new CB();
}
}
```

In this example, we have two arrays of dog names and the processing is quite simple; we just convert the first letter of each dog name from lowercase to uppercase. Once both arrays have been processed, we then combine the results into an `ArrayList` of dog names. In the class `CB`, we define the `result` `ArrayList`, where we'll write results when both threads have finished processing the two arrays, `dogs1` and `dogs2`. We also declare the `CyclicBarrier` that both threads will use to wait for each other. Let's step through how the code executes (follow the numbers in the code):

1. In the `CB` constructor, we create a new `CyclicBarrier`, passing in the number of threads that will wait at the barrier **and a Runnable that will be run by the last thread to reach the barrier**. Here, we're using a lambda expression for this `Runnable`.
2. We then create two threads, `t1` and `t2`, and pass each the `ProcessDogs` `Runnable`. When we create the `ProcessDogs` `Runnables` for the threads, we pass in the array that the thread will process: we pass `dogs1` to thread `t1` and `dogs2` to thread `t2`.
3. We then start both threads and print a message saying the main thread is done.
4. The two threads begin running and process their respective arrays. The `ProcessDogs` `run()` method simply iterates through the array and converts the first letter of the dog name to uppercase, storing the modified string back in the original array.
5. Once the loop is complete, the thread then waits at the barrier by calling the barrier's `await()` method. We'll come back and talk about the exception handling in a minute.
6. Once both threads reach the barrier, then the `Runnable` we passed to the `CyclicBarrier` constructor is executed by the last thread to reach the barrier. This adds all the items from both arrays to the `ArrayList` `result` and prints the `result`.
7. When the `Runnable` completes, then both threads are released and can continue executing where they left off at the barrier. In this example, each thread just prints a message to the console indicating it's done (with the thread name), but you can imagine that in a more realistic example, they could continue on to do more processing. Notice that neither thread

can print out that it's done until *both* threads have reached the barrier and the barrier Runnable is complete.

If you have a machine with at least two cores and you run this code several times, you may see thread 0 displaying the result or you may see thread 1 displaying the result. It just depends on which thread gets to the barrier last. Also, notice that the main thread often finishes before the other threads, but not always. The most important point to notice is that *every time* you run this code, you will see the resulting `ArrayList` displayed *before* the messages from the threads that they are done. Why? Because both threads must wait at the barrier for the barrier Runnable to complete before they can continue!

Main Thread is done

Thread-0 Result: [Boi, Clover, Charis, Aiko, Zooey, Biscuit]

Thread-0 is done!

Thread-1 is done!

Now let's get back to the exception handling on the `barrier.await()` method. What if one of the threads gets stuck? If this happens, then all the other threads waiting at the barrier get an `InterruptedException` or a `BrokenBarrierException` and are all released. In that case, the barrier Runnable will not run and the remaining threads will continue.

You can reuse a `CyclicBarrier` or use multiple `CyclicBarriers` to coordinate threads at more than one barrier point. For example, you might have the thread Runnable execute some code, wait at barrier 1, then execute some more code, wait at barrier 2, and then finally execute more code and finish. To reset a barrier to its initial state, call the `reset()` method. If you need the main thread to also wait at the barrier, you can call `await()` on the barrier in the main thread, but don't forget to increase the number of threads allowed at the barrier by one when you create the `CyclicBarrier`.

CERTIFICATION OBJECTIVE

Use Executors and ThreadPools (OCP Objective 10.1)

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

Executors (and the ThreadPools used by them) help meet two of the same needs as Threads do:

1. Creating and scheduling some Java code for execution and
2. Optimizing the execution of that code for the hardware resources you have available (using all CPUs, for example)

With traditional threading, you handle needs 1 and 2 yourself. With Executors, you handle need 1, but you get to use an off-the-shelf solution for need 2. The `java.util.concurrent` package provides several different off-the-shelf solutions (Executors and ThreadPools), which you'll read about in this chapter.



When you have multiple needs or concerns, it is common to separate the code for each need into different classes. This makes your application more modular and flexible. This is a fundamental programming principle called “separation of concerns.”

In a way, an Executor is an alternative to starting new threads. Using Threads directly can be considered low-level multithreading, whereas using Executors can be considered high-level multithreading. To understand how an Executor can replace manual thread creation, let us first analyze what happens when starting a new thread.

1. First, you must identify a task of some sort that forms a self-contained unit of work. You will typically code this task as a class that implements the `Runnable` interface.
2. After creating a `Runnable`, the next step is to execute it. You have two options for executing a `Runnable`:

- **Option one** Call the `run` method synchronously (i.e., without starting a thread). This is probably **not** what you would normally do.

```
Runnable r = new MyRunnableTask();  
r.run(); // executed by calling thread
```

- **Option two** Call the method indirectly, most likely with a new thread.

```
Runnable r = new MyRunnableTask();  
Thread t1 = new Thread(r);  
t1.start();
```

The second approach has the benefit of executing your task asynchronously, meaning the primary flow of execution in your program can continue executing, without waiting for the task to complete. On a multiprocessor system, you must divide a program into a collection of asynchronous tasks that can execute concurrently in order to take advantage of all of the computing power a system possesses.

Identifying Parallel Tasks

Some applications are easier to divide into separate tasks than others. A single-user desktop application may only have a handful of tasks that are suitable for concurrent execution. Networked multiuser servers, on the other hand, have a natural division of work. Each user's actions can be a task. Continuing our computer game scenario, imagine a computer program that can play chess against thousands of people simultaneously. Each player submits his or her move, the computer calculates its move, and finally it informs the player of that move.

Why do we need an alternative to `new Thread(r).start()`? What are the drawbacks? If we use our online chess game scenario, then having 10,000 concurrent players might mean 10,001 concurrent threads. (One thread awaits network connections from clients and performs a `Thread(r).start()` for each player.) The player thread would be responsible for reading the player's move, computing the computer's move, and making the response.

How Many Threads Can You Run?

Do you own a computer that can concurrently run 10,000 threads or 1,000 or even 100? Probably not—this is a trick question. A quad-core CPU (with four processors per unit) might be able to execute two threads per core for a total of eight concurrently executing threads. You can start 10,000 threads, but not all of them will be running at the same time. The underlying operating system's task scheduler rotates the threads so that they each get a slice of time on a processor.

Ten thousand threads all competing for a turn on a processor wouldn't make for a very responsive system. Threads would either have to wait so long for a turn or get such small turns (or both) that performance would suffer.

In addition, each thread consumes system resources. It takes processor cycles to perform a context switch (saving the state of a thread and resuming another thread), and each thread consumes system memory for its stack space. Stack space is used for temporary storage and to keep track of where a thread returns to after completing a method call. Depending on a thread's behavior, it might be possible to lower the cost (in RAM) of creating a thread by reducing a thread's stack size.



To reduce a thread's stack size, the Oracle JVM supports using the nonstandard-xss1024k option to the java command. Note that decreasing the value too far can result in some threads throwing exceptions when performing certain tasks, such as making a large number of recursive method calls.

Another limiting factor in being able to run 10,000 threads in an application has to do with the underlying limits of the OS. Operating systems typically have limits on the number of threads an application can create. These limits can prevent a buggy application from spawning countless threads and making your system unresponsive. If you have a legitimate need to run 10,000 threads, you will probably have to consult your operating system's documentation to discover possible limits and configuration options.

CPU-Intensive vs. I/O-Intensive Tasks

If you correctly configure your OS and you have enough memory for each thread's stack space plus your application's primary memory (heap), will you be able to run an application with 10,000 threads? It depends.... Remember that your processor can only run a small number of concurrent threads (in the neighborhood of 8 to 16 threads). Yet many network server applications, such as our online chess game, would have traditionally started a new thread for each connected client. A system might be able to run an application with such a high number of threads because most of the threads are not doing anything. More

precisely, in an application like our online chess server, most threads would be blocked waiting on I/O operations such as `InputStream.read` or `OutputStream.write` method calls.

When a thread makes an I/O request using `InputStream.read` and the data to be read isn't already in memory, the calling thread will be put to sleep by the system until the requested data can be loaded. This is much more efficient than keeping the thread on the processor while it has nothing to do. I/O operations are extremely slow when compared to compute operations—reading a sector from a hard drive takes much longer than adding hundreds of numbers. A processor might execute hundreds of thousands, or even millions, of instructions while awaiting the completion of an I/O request. The type of work (either CPU intensive or I/O intensive) a thread will be performing is important when considering how many threads an application can safely run. Imagine your world-class-computer 749chess-playing program takes one minute of processor time (no I/O at all) to calculate each move. In this scenario, it would only take about 16 concurrent players to cause your system to have periods of maximum CPU utilization.



If your tasks will be performing I/O operations, you should be concerned about how increased load (users) might affect scalability. If your tasks perform blocking I/O, then you might need to utilize a thread-per-task model. If you don't, then all your threads may be tied up in I/O operations with no threads remaining to support additional users. Another option would be to investigate whether you can use nonblocking I/O instead of blocking I/O.

Fighting for a Turn

If it takes the computer player one minute to calculate a turn and it takes a human player about the same time, then each player only uses one minute of CPU time out of every two minutes of real time. With a system capable of executing 16 concurrent game threads, that means we could handle 32 connected players. But if all 32 players make their turn at once, the computer will be stuck trying to calculate 32 moves at once. If the system uses preemptive multitasking (the most common type), then each thread will get preempted while it is running

(paused and kicked off the CPU) so a different thread can take a turn (time slice). In most JVM implementations, this is handled by the underlying operating system's task scheduler. The task scheduler is itself a software program. The more CPU cycles spent scheduling and preempting threads, the less processor time you have to execute your application threads. Note that it would appear to the untrained observer that all 32 threads were running concurrently because a preemptive multitasking system will switch out the running threads frequently (millisecond time slices).

Decoupling Tasks from Threads

The best design would be one that utilized as many system resources as possible without attempting to overutilize the system. If 16 threads are all you need to fully utilize your CPU, why would you start more than that? In a traditional system, you start more threads than your system can concurrently run and hope that only a small number are in a running state. If we want to adjust the number of threads that are started, we need to decouple the tasks that are to be performed (our `Runnable` instances) from our thread creation and starting. This is where a `java.util.concurrent.Executor` can help. The basic usage looks something like this:

```
Runnable r = new MyRunnableTask();
Executor ex = // details to follow
ex.execute(r);
```

A `java.util.concurrent.Executor` is used to execute the `run` method in a `Runnable` instance much like a thread. Unlike a more traditional `new Thread(r).start()`, an `Executor` can be designed to use any number of threading approaches, including

- Not starting any threads at all (task is run in the calling thread)
- Starting a new thread for each task
- Queuing tasks and processing them with only enough threads to keep the CPU utilized

You can easily create your own implementations of an `Executor` with custom behaviors. As you'll see soon, several implementations are provided in the standard Java SE libraries. Looking at sample `Executor` implementations can

help you to understand their behavior. This next example doesn't start any new threads; instead, it executes the `Runnable` using the thread that invoked the `Executor`.

```
import java.util.concurrent.Executor;
public class SameThreadExecutor implements Executor {
    @Override
    public void execute(Runnable command) {
        command.run(); // caller waits
    }
}
```

The following `Executor` implementation would use a new thread for each task:

```
import java.util.concurrent.Executor;
public class NewThreadExecutor implements Executor {
    @Override
    public void execute(Runnable command) {
        Thread t = new Thread(command);
        t.start();
    }
}
```

This example shows how an `Executor` implementation can be put to use:

```
Runnable r = new MyRunnableTask();
Executor ex = new NewThreadExecutor(); // choose Executor
ex.execute(r);
```

By coding to the `Executor` interface, the submission of tasks is decoupled from the execution of tasks. The result is that you can easily modify how threads

are used to execute tasks in your applications.



There is no “right number” of threads for task execution. The type of task (CPU intensive versus I/O intensive), number of tasks, I/O latency, and system resources all factor into determining the ideal number of threads to use. You should test your applications to determine the ideal threading model. This is one reason why the ability to separate task submission from task execution is important.

Several Executor implementations are supplied as part of the standard Java libraries. The Executors class (notice the “s” at the end) is a factory for Executor implementations.

```
Runnable r = new MyRunnableTask();  
Executor ex = Executors.newCachedThreadPool(); // choose Executor  
ex.execute(r);
```

The Executor instances returned by Executors are actually of type ExecutorService (which extends Executor). An ExecutorService provides management capability and can return Future instances that are used to obtain the result of executing a task asynchronously. We’ll talk more about Future in a few pages!

```
Runnable r = new MyRunnableTask();  
ExecutorService ex = Executors.newCachedThreadPool(); // subtype of Executor  
ex.execute(r);
```

Three types of ExecutorService instances can be created by the factory methods in the Executors class: cached thread pool executors, fixed thread pool executors, and single thread pool executors.

Cached Thread Pools

```
ExecutorService ex = Executors.newCachedThreadPool();
```

A cached thread pool will create new threads as they are needed and reuse threads that have become free. Threads that have been idle for 60 seconds are removed from the pool.

Watch out! Without some type of external limitation, a cached thread pool may be used to create more threads than your system can handle.

Fixed Thread Pools—Most Common

```
ExecutorService ex = Executors.newFixedThreadPool(4);
```

A fixed thread pool is constructed using a numeric argument (4 in the preceding example) that specifies the number of threads used to execute tasks. This type of pool will probably be the one you use the most because it prevents an application from overloading a system with too many threads. Tasks that cannot be executed immediately are placed on an unbounded queue for later execution.



You might base the number of threads in a fixed thread pool on some attribute of the system your application is executing on. By tying the number of threads to system resources, you can create an application that scales with changes in system hardware. To query the number of available processors, you can use the java.lang.Runtime class.

```
Runtime rt = Runtime.getRuntime();
int cpus = rt.availableProcessors();
```

ThreadPoolExecutor

Both `Executors.newCachedThreadPool()` and `Executors.newFixedThreadPool(4)` return objects of type `java.util.concurrent.ThreadPoolExecutor` (which implements `ExecutorService` and `Executor`). You will typically use the `Executors` factory methods instead of creating `ThreadPoolExecutor` instances directly, but you can

cast the fixed or cached thread pool `ExecutorService` references if you need access to the additional methods. The following example shows how you could dynamically adjust the thread count of a pool at runtime:

```
ThreadPoolExecutor tpe = (ThreadPoolExecutor)Executors.newFixedThreadPool(4);  
tpe.setCorePoolSize(8);  
tpe.setMaximumPoolSize(8);
```

Single Thread Pools

```
ExecutorService ex = Executors.newSingleThreadExecutor();
```

A single thread pool uses a single thread to execute tasks. Tasks that cannot be executed immediately are placed on an unbounded queue for later execution. Unlike a fixed thread pool executor with a size of 1, a single thread executor prevents any adjustments to the number of threads in the pool.

Scheduled Thread Pool

In addition to the three basic `ExecutorService` behaviors outlined already, the `Executors` class has factory methods to produce a `ScheduledThreadPoolExecutor`. A `ScheduledThreadPoolExecutor` enables tasks to be executed after a delay or at repeating intervals. Here, we see some thread-scheduling code in action:

```

ScheduledExecutorService ftses =
    Executors.newScheduledThreadPool(4);           // multi-threaded
                                                // version
    ftses.schedule(r, 5, TimeUnit.SECONDS);        // run once after
                                                // a delay
    ftses.scheduleAtFixedRate(r, 2, 5, TimeUnit.SECONDS); // begin after a
                                                // 2sec delay
                                                // and begin again every 5 seconds
    ftses.scheduleWithFixedDelay(r, 2, 5, TimeUnit.SECONDS); // begin after
                                                // 2sec delay
                                                // and begin again 5 seconds *after* completing the last execution

```

The Callable Interface

So far, the Executors examples have used a Runnable instance to represent the task to be executed. The `java.util.concurrent.Callable` interface serves the same purpose as the Runnable interface, but provides more flexibility. Unlike the Runnable interface, a Callable may return a result upon completing execution and may throw a checked exception. An `ExecutorService` can be passed a Callable instead of a Runnable.



Avoid using methods such as `Object.wait`, `Object.notify`, and `Object.notifyAll` in tasks (Runnable and callable instances) that are submitted to an Executor or ExecutorService. Because you might not know what the threading behavior of an Executor is, it is a good idea to avoid operations that may interfere with thread execution. Avoiding these types of methods is advisable anyway since they are easy to misuse.

The primary benefit of using a Callable is the ability to return a result. Because an `ExecutorService` may execute the Callable asynchronously (just

like a `Runnable`), you need a way to check the completion status of a `Callable` and obtain the result later. A `java.util.concurrent.Future` is used to obtain the status and result of a `Callable`. Without a `Future`, you'd have no way to obtain the result of a completed `Callable` and you might as well use a `Runnable` (which returns `void`) instead of a `Callable`. Here's a simple `Callable` example that loops a random number of times and returns the random loop count:

```
import java.util.concurrent.Callable;
import java.util.concurrent.ThreadLocalRandom;
public class MyCallable implements Callable<Integer> {

    @Override
    public Integer call() {
        // Obtain a random number from 1 to 10
        int count = ThreadLocalRandom.current().nextInt(1, 11);
        for(int i = 1; i <= count; i++) {
            System.out.println("Running..." + i);
        }
        return count;
    }
}
```

Submitting a `Callable` to an `ExecutorService` returns a `Future` reference. When you use the `Future` to obtain the `Callable`'s result, you will have to handle two possible exceptions:

- **InterruptedException** Raised when the thread calling the `Future`'s `get()` method is interrupted before a result can be returned
- **ExecutionException** Raised when an exception was thrown during the execution of the `Callable`'s `call()` method

```

Callable<Integer> c = new MyCallable();
ExecutorService ex =
    Executors.newCachedThreadPool();
Future<Integer> f = ex.submit(c); // finishes in the future
try {
    Integer v = f.get(); // blocks until done
    System.out.println("Ran:" + v);
} catch (InterruptedException | ExecutionException iex) {
    System.out.println("Failed");
}

```



I/O activities in your `Runnable` and `Callable` instances can be a serious bottleneck. In preceding examples, the use of `System.out.println()` will cause I/O activity. If this wasn't a trivial example being used to demonstrate `Callable` and `ExecutorService`, you would probably want to avoid repeated calls to `println()` in the `Callable`. One possibility would be to use `StringBuilder` to concatenate all output strings and have a single `println()` call before the `call()` method returns. Another possibility would be to use a logging framework (see `java.util.logging`) in place of any `println()` calls.

ThreadLocalRandom

The first `Callable` example used a `java.util.concurrent.ThreadLocalRandom`. `ThreadLocalRandom` was introduced in Java 7 as a new way to create random numbers. `Math.random()` and shared `Random` instances are thread-safe, but suffer from contention when used by multiple threads. A `ThreadLocalRandom` is unique to a thread and will perform better because it avoids any contention. `ThreadLocalRandom` also provides several convenient methods such as `nextInt(int, int)` that allow you to specify the range of possible values returned.

ExecutorService Shutdown

You've seen how to create Executors and how to submit Runnable and Callable tasks to those Executors. The final component to using an Executor is shutting it down once it is done processing tasks. An ExecutorService should be shut down once it is no longer needed to free up system resources and to allow graceful application shutdown. Because the threads in an ExecutorService may be nondaemon threads, they may prevent normal application termination. In other words, your application stays running after completing its main method. You could perform a `System.exit(0)` call, but it would preferable to allow your threads to complete their current activities (especially if they are writing data).

```
ExecutorService ex =
// ...
ex.shutdown(); // no more new tasks
                // but finish existing tasks
try {
    boolean term = ex.awaitTermination(2, TimeUnit.SECONDS);
    // wait 2 seconds for running tasks to finish
} catch (InterruptedException ex1) {
    // did not wait the full 2 seconds
} finally {
    if(!ex.isTerminated()) // are all tasks done?
    {
        List<Runnable> unfinished = ex.shutdownNow();
        // a collection of the unfinished tasks
    }
}
```

For long-running tasks (especially those with looping constructs), consider using `Thread.currentThread().isInterrupted()` to determine if a Runnable

or Callable should return early. The ExecutorService.shutdownNow() method will typically call Thread.interrupt() in an attempt to terminate any unfinished tasks.

CERTIFICATION OBJECTIVE

Use the Parallel Fork/Join Framework (OCP Objective 10.5)

10.5 *Use the parallel Fork/Join Framework.*

The Fork-Join Framework provides a highly specialized ExecutorService. The other ExecutorService instances you've seen so far are centered on the concept of submitting multiple tasks to an ExecutorService. By doing this, you provide an easy avenue for an ExecutorService to take advantage of all the CPUs in a system by using threads to complete tasks. Sometimes, you don't have multiple tasks; instead, you have one really big task.

There are many large tasks or problems you might need to solve in your application. For example, you might need to initialize the elements of a large array with values. You might think that initializing an array doesn't sound like a large complex task in need of a framework. The key is that it needs to be a **large** task. What if you need to fill up a 100,000,000-element array with randomly generated values? The Fork/Join Framework makes it easier to tackle big tasks like this, while leveraging all of the CPUs in a system.

Divide and Conquer

Certain types of large tasks can be split up into smaller subtasks; those subtasks might, in turn, be split up into even smaller tasks. There is no limit to how many times you might subdivide a task. For example, imagine the task of having to repaint a single long fence that borders several houses. The "paint the fence" task could be subdivided so that each household would be responsible for painting a section of the fence. Each household could then subdivide their section into subsections to be painted by individual family members. In this example, there are three levels of recursive calls. The calls are considered recursive because at each step we are trying to accomplish the same thing: paint the fence. In other words, Joe, one of the home owners, was told by his wife,

“paint that (huge) fence; it looks old.” Joe decides that painting the whole fence is too much work and talks all the households along the fence into taking a subsection. Now Joe is telling himself “paint that (subsection of) fence; it looks old.” Again, Joe decides that it is still too much work and subdivides his section into smaller sections for each member of his household. Again, Joe tells himself “paint that (subsection of) fence; it looks old,” but this time, he decides that the amount of work is manageable and proceeds to paint his section of fence. Assuming everyone else paints their subsections (hopefully in a timely fashion), the result is the entire fence being painted.



When using the Fork/Join Framework, your tasks will be coded to decide how many levels of recursion (how many times to subdivide) are appropriate. You'll want to split things up into enough subtasks that you have adequate tasks to keep all of your CPUs utilized. Sometimes, the best number of tasks can be a little hard to determine because of factors we will discuss later. You might have to benchmark different numbers of task divisions to find the optimal number of subtasks that should be created.

Just because you can use Fork/Join to solve a problem doesn't always mean you should. If our initial task is to paint eight fence planks, then Joe might just decide to paint them himself. The effort involved in subdividing the problem and assigning those tasks to workers (threads) can sometimes be more than the actual work you want to perform. The number of elements (or fence planks) is not the only thing to consider—the amount of work performed on each element is also important. Imagine if Joe was asked to paint a mural on each fence plank. Because processing each element (fence plank) is so time consuming, in this case, it might be beneficial to adopt a divide-and-conquer solution even though there is a small number of elements.

ForkJoinPool

The Fork/Join ExecutorService implementation is `java.util.concurrent.ForkJoinPool`. You will typically submit a single task to a `ForkJoinPool` and await its completion. The `ForkJoinPool` and the task itself work together to divide and conquer the problem. Any problem that can be

recursively divided can be solved using Fork/Join. Anytime you want to perform the same operation on a collection of elements (painting thousands of fence planks or initializing 100,000,000 array elements), consider using Fork/Join.

To create a `ForkJoinPool`, simply call its no-arg constructor:

```
ForkJoinPool fjPool = new ForkJoinPool();
```

The no-arg `ForkJoinPool` constructor creates an instance that will use the `Runtime.availableProcessors()` method to determine the level of parallelism. The level of parallelism determines the number of threads that will be used by the `ForkJoinPool`.

There is also a `ForkJoinPool(int parallelism)` constructor that allows you to override the number of threads that will be used.

ForkJoinTask

Just as with Executors, you must capture the task to be performed as Java code. With the Fork/Join Framework, a `java.util.concurrent.ForkJoinTask` instance (actually a subclass—more on that later) is created to represent the task that should be accomplished. This is different from other executor services that primarily used either `Runnable` or `Callable`. A `ForkJoinTask` concrete subclass has many methods (most of which you will never use), **but the following methods are important: `compute()`, `fork()`, and `join()`.**

A `ForkJoinTask` subclass is where you will perform most of the work involved in completing a Fork/Join task. `ForkJoinTask` is an abstract base class; we will discuss the two subclasses, `RecursiveTask` and `RecursiveAction`, later. The basic structure of any `ForkJoinTask` is shown in this pseudocode example:

```

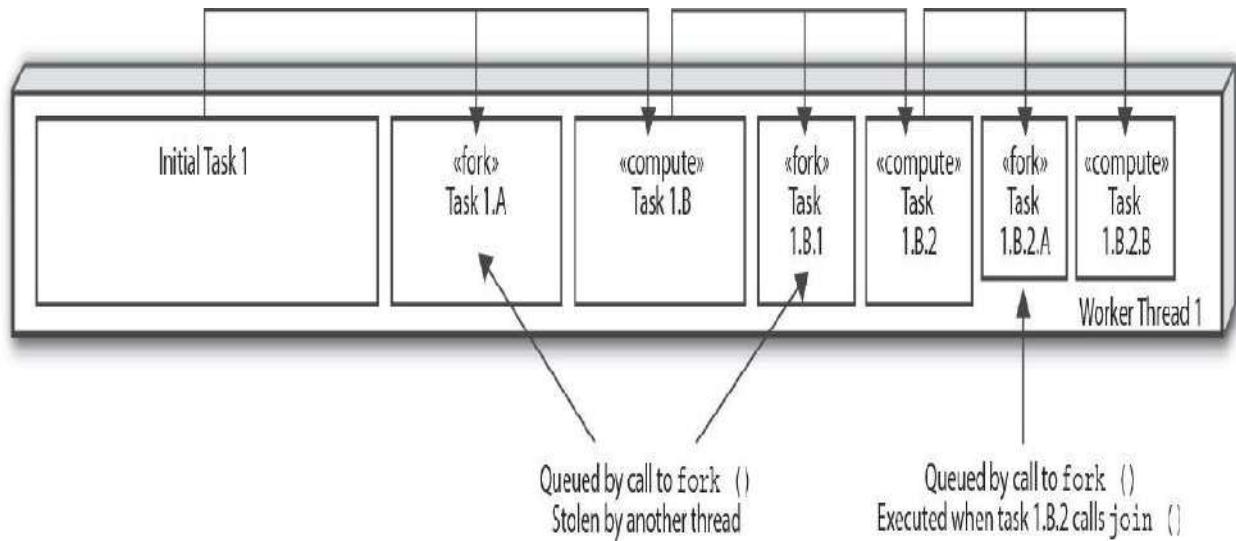
class ForkJoinPaintTask {
    compute() {
        if(isFenceSectionSmall()) { // is it a manageable amount of work?
            paintFenceSection(); // do the task
        } else { // task too big, split it
            ForkJoinPaintTask leftHalf = getLeftHalfOfFence();
            leftHalf.fork(); // queue left half of task
            ForkJoinPaintTask rightHalf = getRightHalfOfFence();
            rightHalf.compute(); // work on right half of task
            leftHalf.join(); // wait for queued task to be complete
        }
    }
}

```

Fork

With the Fork/Join Framework, each thread in the ForkJoinPool has a queue of the tasks it is working on; this is unlike most ExecutorService implementations that have a single shared task queue. The `fork()` method places a `ForkJoinTask` in the current thread's task queue. A normal thread does not have a queue of tasks—only the specialized threads in a `ForkJoinPool` do. This means that you can only call `fork()` if you are within a `ForkJoinTask` that is being executed by a `ForkJoinPool`.

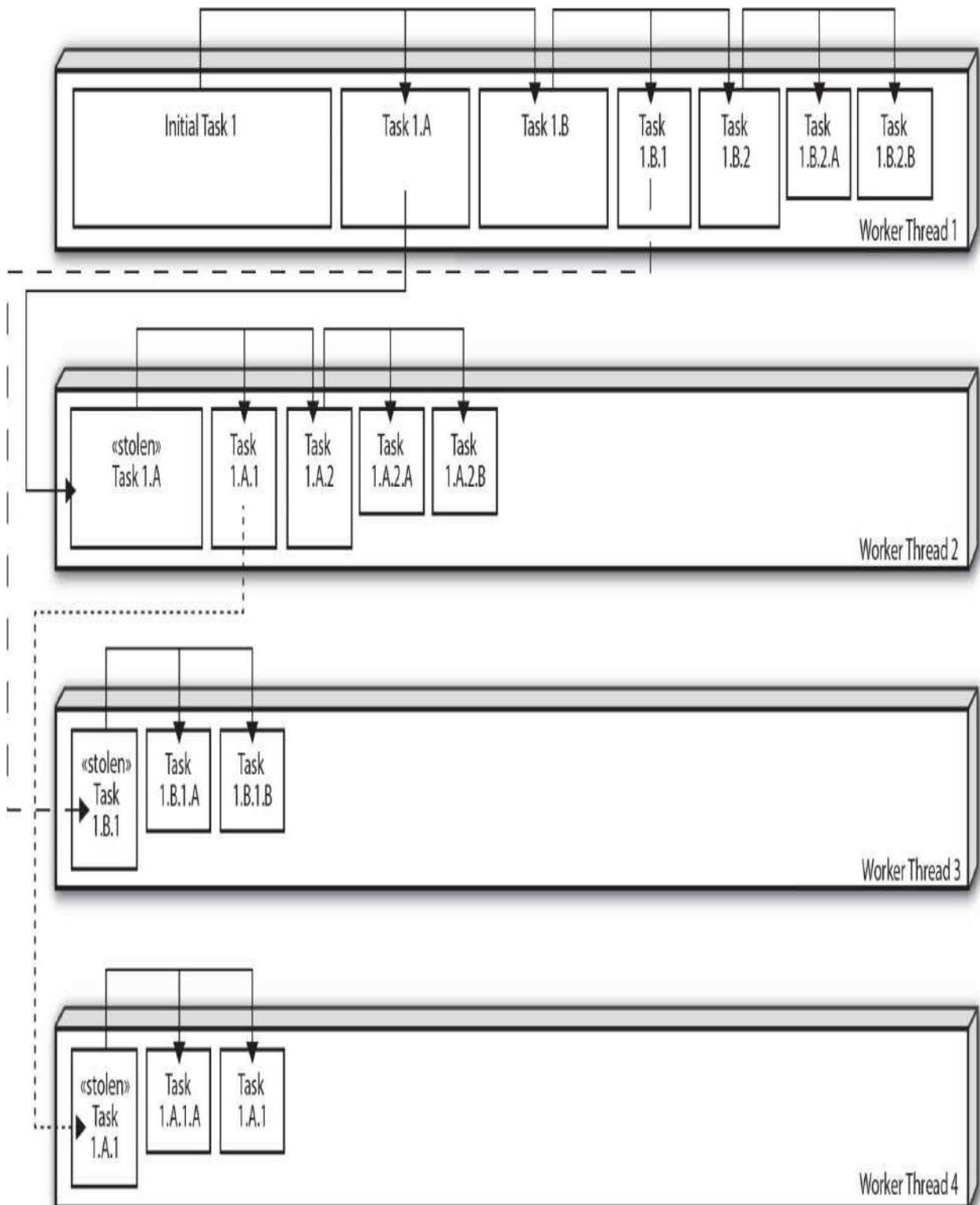
Initially, only a single thread in a `ForkJoinPool` will be busy when you submit a task. That thread will begin to subdivide the tasks into smaller tasks. Each time a task is subdivided into two subtasks, you fork (or queue) the first task and compute the second task. In the event you need to subdivide a task into more than two subtasks, each time you split a task, you would fork every new subtask except one (which would be computed).



Work Stealing

Notice how the call to `fork()` is placed before the call to `compute()` or `join()`. A key feature of the Fork/Join Framework is work stealing. Work stealing is how the other threads in a `ForkJoinPool` will obtain tasks. When initially submitting a Fork/Join task for execution, a single thread from a `ForkJoinPool` begins executing (and subdividing) that task. Each call to `fork()` places a new task in the calling thread's task queue. The order in which the tasks are queued is important. The tasks that have been queued the longest represent larger amounts of work. In the `ForkJoinPaintTask` example, the task that represents 100 percent of the work would begin executing, and its first queued (forked) task would represent 50 percent of the fence, the next 25 percent, then 12.5 percent, and so on. Of course, this can vary, depending on how many times the task will be subdivided and whether we are splitting the task into halves or quarters or some other division, but in this example, we are splitting each task into two parts: queuing one part and executing the second part.

The nonbusy threads in a `ForkJoinPool` will attempt to steal the oldest (and, therefore, largest) task from any Fork/Join thread with queued tasks. Given a `ForkJoinPool` with four threads, one possible sequence of events could be that the initial thread queues tasks that represent 50 percent and 25 percent of the work, which are then stolen by two different threads. The thread that stole the 50 percent task then subdivides that task and places a 25 percent task on its queue, which is then stolen by a fourth thread, resulting in four threads that each process 25 percent of the work.



Of course, if everything was always this evenly distributed, you might not have as much of a need for Fork/Join. You could just presplit the work into a number of tasks equal to the number of threads in your system and use a regular

`ExecutorService`. In practice, each of the four threads will not finish their 25 percent of the work at the same time—one thread will be the slow thread that doesn’t get as much work done. There are many reasons for this: The data being processed may affect the amount of computation (25 percent of an array might not mean 25 percent of the workload), or a thread might not get as much time to execute as the other threads. Operating systems and other running applications are also going to consume CPU time. In order to finish executing the Fork/Join task as soon as possible, the threads that finish their portions of the work first will start to steal work from the slower threads—this way, you will be able to keep all of the CPU involved. If you only split the tasks into 25 percent of the data (with four threads), then there would be nothing for the faster threads to steal from when they finish early. In the beginning, if the slower thread stole 25 percent of the work and started processing it without further subdividing and queuing, then there would be no work on the slow thread’s queue to steal. You should subdivide the tasks into a few more sections than are needed to evenly distribute the work among the number of threads in your `ForkJoinPools` because threads will most likely not perform exactly the same. Subdividing the tasks is extra work—if you do it too much, you might hurt performance. Subdivide your tasks enough to keep all CPUs busy, but not more than is needed. Unfortunately, there is no magic number to split your tasks into—it varies based on the complexity of the task, the size of the data, and even the performance characteristics of your CPUs.

Back to fence painting, make the `isFenceSectionSmall()` logic as simple as possible (low overhead) and easy to change. You should benchmark your Fork/Join code (using the hardware that you expect the code to typically run on) and find an amount of task subdivision that works well. It doesn’t have to be perfect; once you are close to the ideal range, you probably won’t see much variation in performance unless other factors come into play (different CPUs, etc.).

Join

When you call `join()` on the (left) task, it should be one of the last steps in the `compute` method, after calling `fork()` and `compute()`. Calling `join()` says, “I can only proceed when this (left) task is done.” Several possible things can happen when you call `join()`:

- The task you call `join()` on might already be done. Remember you are calling `join()` on a task that already had `fork()` called. The task might

have been stolen and completed by another thread. In this case, calling `join()` just verifies the task is complete and you can continue on.

- The task you call `join()` on might be in the middle of being processed. Another thread could have stolen the task, and you'll have to wait until the joined task is done before continuing.
- The task you call `join()` on might still be in the queue (not stolen). In this case, the thread calling `join()` will execute the joined task.

RecursiveAction

`ForkJoinTask` is an abstract base class that outlines most of the methods, such as `fork()` and `join()`, in a Fork/Join task. If you need to create a `ForkJoinTask` that does not return a result, then you should subclass `RecursiveAction`.

`RecursiveAction` extends `ForkJoinTask` and has a single abstract `compute` method that you must implement:

```
protected abstract void compute();
```

An example of a task that does not need to return a result would be any task that initializes an existing data structure. The following example will initialize an array to contain random values. Notice that there is only a single array throughout the entire process. When subdividing an array, you should avoid creating new objects when possible.

```
public class RandomInitRecursiveAction extends RecursiveAction {  
    private static final int THRESHOLD = 10000;  
    private int[] data;  
    private int start;  
    private int end;  
  
    public RandomInitRecursiveAction(int[] data, int start, int end) {  
        this.data = data;  
        this.start = start; // where does our section begin?  
        this.end = end; // how large is this section?  
    }  
    @Override  
    protected void compute() {  
        if (end - start <= THRESHOLD) { // is it a manageable amount of work?  
            // do the task  
            for (int i = start; i < end; i++) {  
                data[i] = ThreadLocalRandom.current().nextInt();  
            }  
        } else { // task too big, split it  
            int halfWay = ((end - start) / 2) + start;  
            RandomInitRecursiveAction a1 =  
                new RandomInitRecursiveAction(data, start, halfWay);  
            a1.fork(); // queue left half of task  
            RandomInitRecursiveAction a2 =  
                new RandomInitRecursiveAction(data, halfWay, end);  
        }  
    }  
}
```

```

        a2.compute();      // work on right half of task
        a1.join();        // wait for queued task to be complete
    }
}
}

```

Sometimes, you will see one of the `invokeAll` methods from the `ForkJoinTask` class used in place of the `fork/compute/join` method combination. 763The `invokeAll` methods are convenience methods that can save some typing. Using them will also help you avoid bugs! The first task passed to `invokeAll` will be executed (`compute` is called), and all additional tasks will be forked and joined. In the preceding example, you could eliminate the three `fork/compute/join` lines and replace them with a single line:

```
invokeAll(a2, a1);
```

To begin the application, we create a large array and initialize it using Fork/Join:

```

public static void main(String[] args) {
    int[] data = new int[10_000_000];
    ForkJoinPool fjPool = new ForkJoinPool();
    RandomInitRecursiveAction action =
        new RandomInitRecursiveAction(data, 0, data.length);
    fjPool.invoke(action);
}

```

Notice that we do not expect any return values when calling `invoke`. A `RecursiveAction` returns nothing.

RecursiveTask

If you need to create a `ForkJoinTask` that does return a result, then you should subclass `RecursiveTask`. `RecursiveTask` extends `ForkJoinTask` and has a single

abstract compute method that you must implement:

```
protected abstract V compute(); // V is a generic type
```

The following example will find the position in an array with the greatest value; if duplicate values are found, the first occurrence is returned. Notice that there is only a single array throughout the entire process. (Just like before, when subdividing an array, you should avoid creating new objects when possible.)

```
public class FindMaxPositionRecursiveTask extends RecursiveTask<Integer> {  
    private static final int THRESHOLD = 10000;  
    private int[] data;  
    private int start;  
    private int end;
```

```

public FindMaxPositionRecursiveTask(int[] data, int start, int end) {
    this.data = data;
    this.start = start;
    this.end = end;
}

@Override
protected Integer compute() {      // return type matches the <generic> type
    if (end - start <= THRESHOLD) { // is it a manageable amount of work?
        int position = 0;          // if all values are equal, return
                                    // position 0
        for (int i = start; i < end; i++) {
            if (data[i] > data[position]) {
                position = i;
            }
        }
        return position;
    } else { // task too big, split it
        int halfWay = ((end - start) / 2) + start;
        FindMaxPositionRecursiveTask t1 =
            new FindMaxPositionRecursiveTask(data, start, halfWay);
        t1.fork(); // queue left half of task
        FindMaxPositionRecursiveTask t2 =
            new FindMaxPositionRecursiveTask(data, halfWay, end);
        int position2 = t2.compute(); // work on right half of task
        int position1 = t1.join();   // wait for queued task to be complete
        // out of the position in two subsection which is greater?
        if (data[position1] > data[position2]) {
            return position1;
        } else if (data[position1] < data[position2]) {
            return position2;
        } else {
            return position1 < position2 ? position1 : position2;
        }
    }
}
}

```

To begin the application, we reuse the `RecursiveAction` example to create a large array and initialize it using Fork/Join. After initializing the array with random values, we reuse the `ForkJoinPool` with our `RecursiveTask` to find the position with the greatest value:

```
public static void main(String[] args) {  
    int[] data = new int[10_000_000];  
    ForkJoinPool fjPool = new ForkJoinPool();  
    RandomInitRecursiveAction action =  
        new RandomInitRecursiveAction(data, 0, data.length);  
    fjPool.invoke(action);  
    // new code begins here  
  
    FindMaxPositionRecursiveTask task =  
        new FindMaxPositionRecursiveTask(data, 0, data.length);  
    Integer position = fjPool.invoke(task);  
    System.out.println("Position: " + position + ", value: " + data[position]);  
}
```

Notice that a value is returned by the call to `invoke` when using a `RecursiveTask`.



If your application will repeatedly submit tasks to a `ForkJoinPool`, then you should reuse a single `ForkJoinPool` instance and avoid the overhead involved in creating a new instance.

Embarrassingly Parallel

A problem or task is said to be embarrassingly parallel if little or no additional

work is required to solve the problem in a parallel fashion. Sometimes, solving a problem in parallel adds so much more overhead that the problem can be solved faster serially. The `RandomInitRecursiveAction` example, which initializes an array to random values, has no additional overhead because what happens when processing one subsection of an array has no bearing on the processing of another subsection. Technically, there is a small amount of overhead even in the `RandomInitRecursiveAction`; the Fork/Join Framework and the `if` statement that determines whether the problem should be subdivided both introduce some overhead. Be aware that it can be difficult to get performance gains that scale with the number of CPUs you have. Typically, four CPUs will result in less than a 4× speedup when moving from a serial to a parallel solution.

The `FindMaxPositionRecursiveTask` example, which finds the largest value in an array, does introduce a small additional amount of work because you must compare the result from each subsection and determine which is greater. This is only a small amount, however, and adds little overhead. Some tasks may introduce so much additional work that any advantage of using parallel processing is eliminated (the task runs slower than serial execution). If you find yourself performing a lot of processing after calling `join()`, then you should benchmark your application to determine if there is a performance benefit to using parallel processing. Be aware that performance benefits might only be seen with a certain number of CPUs. A task might run on one CPU in 5 seconds, on two CPUs in 6 seconds, and on four CPUs in 3.5 seconds.

The Fork/Join Framework is designed to have minimal overhead as long as you don't over-subdivide your tasks and the amount of work required to join results can be kept small. A good example of a task that incurs additional overhead but still benefits from Fork/Join is array sorting. When you split an array into two halves and sort each half separately, you then have to combine the two sorted arrays, as shown in the following example:

```
public class SortRecursiveAction extends RecursiveAction {
    private static final int THRESHOLD = 1000;
    private int[] data;
    private int start;
    private int end;

    public SortRecursiveAction(int[] data, int start, int end) {
        this.data = data;
        this.start = start;
        this.end = end;
    }

    @Override
    protected void compute() {
        if (end - start <= THRESHOLD) {
            Arrays.sort(data, start, end);
        } else {
            int halfWay = ((end - start) / 2) + start;
            SortRecursiveAction a1 =
                new SortRecursiveAction(data, start, halfWay);
            SortRecursiveAction a2 =
                new SortRecursiveAction(data, halfWay, end);
            invokeAll(a1, a2); // shortcut for fork() & join()
            if (data[halfWay-1] <= data[halfWay]) {
                return; // already sorted
            }
            // merging of sorted subsections begins here
            int[] temp = new int[end - start];
            int s1 = start, s2 = halfWay, d = 0;
            while (s1 < halfWay && s2 < end) {
                if (data[s1] < data[s2]) {
                    temp[d++] = data[s1++];
                } else if (data[s1] > data[s2]) {
                    temp[d++] = data[s2++];
                } else {
                    temp[d++] = data[s1++];
                    temp[d++] = data[s2++];
                }
            }
            if (s1 != halfWay) {
                System.arraycopy(data, s1, temp, d, temp.length - d);
            }
        }
    }
}
```

```
        } else if(s2 != end) {
            System.arraycopy(data, s2, temp, d, temp.length - d);
        }
        System.arraycopy(temp, 0, data, start, temp.length);
    }
}
```

In the previous example, everything after the call to `invokeAll` is related to merging two sorted subsections of an array into a single larger sorted subsection.



Because Java applications are portable, the system running your application may not have the hardware resources required to see a performance benefit. Always perform testing to determine which problem and hardware combinations see performance increases when using Fork/Join.

CERTIFICATION OBJECTIVE

Parallel Streams (OCP Objective 10.6)

10.6 Use parallel Streams including reduction, decomposition, merging processes, pipelines and performance.

Parallel streams are designed for problems you can divide and conquer, just like the problems described in the previous section. In fact, parallel streams are implemented with Fork/Join tasks under the covers, so you’re essentially doing the same thing: that is, splitting up a problem into subtasks that can be executed on separate threads and then joining them back together to produce a result.

Unlike the code you write to take advantage of the Fork/Join Framework, parallel stream code is relatively easy to write. If you think writing

`RecursiveActions` and `RecursiveTasks` is tricky (we do too!), you'll be pleased with how much easier parallel streams can be. That said, there are several gotchas to watch out for when using parallel streams, so even though the code is easier to write, you need to pay close attention.

And, like we said about using the Fork/Join Framework, just because you can use parallel streams to solve a problem doesn't always mean you should. The same concerns apply: the effort to create tasks that can run in threads can add enough overhead that using parallel streams is sometimes slower than using sequential streams. We'll do testing as we work through some parallel stream code and check our results as we go to make sure we get the performance gains we expect.

How to Make a Parallel Stream Pipeline

In [Chapter 9](#), “Streams,” we made a stream parallel by calling the `parallel()` method on the stream, like this:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int sum = nums.stream()
    .parallel()          // make the stream parallel
    .mapToInt(n -> n)
    .sum();
System.out.println("Sum is: " + sum);
```

Here, we take a list of numbers, stream them, map each `Integer` to an `int`, and sum. To make this very simple stream pipeline parallel, we simply call the `parallel()` method on the stream.

Let's take a peek at how the tasks created by this parallel stream get split up and handled by workers in the `ForkJoinPool`. We can do that by adding a `peek()` into the pipeline and then printing the name of the thread handling each `Integer` in the stream:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int sum = nums.stream()
    .parallel()          // make the stream parallel
    .peek(i ->         // print the thread for the worker
        System.out.println(i + ": "
            + Thread.currentThread().getName()))
    .mapToInt(n -> n)
    .sum();
System.out.println("Sum is: " + sum);
```

The output (on our computer) is

```
7: main
5: ForkJoinPool.commonPool-worker-5
8: ForkJoinPool.commonPool-worker-4
9: ForkJoinPool.commonPool-worker-2
3: ForkJoinPool.commonPool-worker-1
1: ForkJoinPool.commonPool-worker-6
10: ForkJoinPool.commonPool-worker-7
2: ForkJoinPool.commonPool-worker-3
6: main
4: ForkJoinPool.commonPool-worker-5
Sum is: 55
```

You will likely see different output (although the same final sum) because your computer may have a different number of cores, and the tasks will likely get split up differently. We ran this code on a computer with eight cores, and you can see that the computer used all eight. We might not actually want the computer to use all eight cores (especially for this simple sum), so we can tell the computer exactly how many workers to use by creating a custom

`ForkJoinPool` and then submitting a task to the pool for execution:

```
ForkJoinPool fjp = new ForkJoinPool(2);
try {
    int sum =
        fjp.submit(                  // returns a Future (FutureTask)
            () -> nums.stream()   // a Callable (value returning task)
                .parallel()       // make the stream parallel
                .peek(i ->
                    System.out.println(i + ": " +
                        Thread.currentThread().getName()))
                .mapToInt(n -> n)
                .sum()
        ).get();                 // from Future; get() waits for
                                // computation to complete and
                                // gets the result
    System.out.println("FJP with 2 workers, sum is: " + sum);
} catch (Exception e) {
    System.out.println("Error executing stream sum");
    e.printStackTrace();
}
```

Here is our output now:

```
7: ForkJoinPool-1-worker-1
6: ForkJoinPool-1-worker-1
9: ForkJoinPool-1-worker-1
3: ForkJoinPool-1-worker-0
10: ForkJoinPool-1-worker-1
5: ForkJoinPool-1-worker-0
8: ForkJoinPool-1-worker-1
4: ForkJoinPool-1-worker-0
2: ForkJoinPool-1-worker-1
1: ForkJoinPool-1-worker-0
FJP with 2 workers, sum is: 55
```

We get the same sum, 55, but now we're using only two workers to do it. This is friendlier to the computer, which may want some cores to do other things while this task is running.

When we call the `submit()` method of the `ForkJoinPool`, we use this method:

```
<T> ForkJoinTask<T> submit(Callable<T> task)
```

and pass a lambda expression for the `Callable`. `Callable` is a functional interface, which is why we can express an instance of a class implementing that interface with the lambda expression. The functional method in `Callable` is `call()`, which "computes a result" (thus, the lambda we are supplying for the `Callable` is a `Supplier`: it takes no arguments and supplies a result).

You've seen how to create a parallel stream with `parallel()`; you can also combine the methods `stream()` and `parallel()` into one method, `parallelStream()`, like this:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int sum = nums.parallelStream()          // make a parallel stream
    .peek(i -> System.out.println(i + ": "
        + Thread.currentThread().getName()))
    .mapToInt(n -> n)
    .sum();
System.out.println("Sum is: " + sum);
```

When we run this code (similar to the first version of the code above), we get the following output:

```
2: ForkJoinPool.commonPool-worker-3
3: ForkJoinPool.commonPool-worker-1
9: ForkJoinPool.commonPool-worker-2
7: main
8: ForkJoinPool.commonPool-worker-4
1: ForkJoinPool.commonPool-worker-6
5: ForkJoinPool.commonPool-worker-5
10: ForkJoinPool.commonPool-worker-7
6: ForkJoinPool.commonPool-worker-1
4: ForkJoinPool.commonPool-worker-3
Sum is: 55
```

Notice the ordering of the processing: the ordering is totally different from the first example above! When summing numbers, it doesn't matter in what order they are summed; we'll get the same result every time. In [Chapter 9](#), you saw that if you display the result of a parallel stream pipeline, the ordering is not guaranteed, and you can see that mixed-up ordering in the output here, too.

You can check to see if a stream is parallel with the `isParallel()` method. This might come in handy if someone hands you a stream and you're not sure. For instance, the following code checks to see if `numsStream` is parallel:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
Stream<Integer> numsStream =           // create a parallel stream
    nums.parallelStream();
// Later...
System.out.println("Is numsStream a parallel stream?? "
    + numsStream.isParallel());
```

You should see the output:

```
Is numsStream a parallel stream?? true
```

What if you have a parallel stream and you want to make it not parallel (i.e., sequential)? You can do that with the `sequential()` method:

```
Stream<Integer> numsStreamSeq =      // make stream sequential
    numsStream.sequential();
System.out.println("Is numsStreamSeq a parallel stream?? "
    + numsStreamSeq.isParallel());
```

and get the output:

```
Is numsStreamSeq a parallel stream?? false
```

Note here that `parallel()`, `isParallel()`, and `sequential()` are methods of `BaseStream` (which is the superinterface of `Stream`), and `parallelStream()` is a method of the collection interface. [Table 11-1](#) summarizes the methods related to parallel streams that you are expected to know for the exam, which we'll cover in this chapter.

TABLE 11-1 Methods Related to Parallel Streams

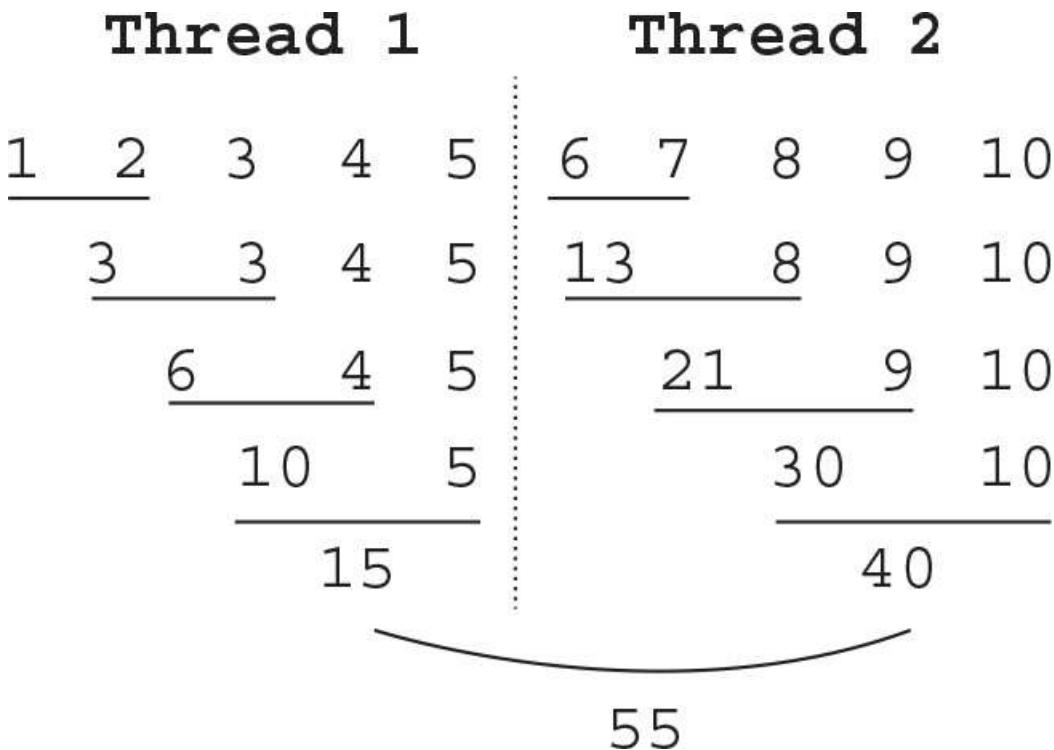
Method	Of Interface	Description
parallel()	BaseStream	Creates a parallel stream from a stream
parallelStream()	Collection	Creates a parallel stream from the Collection source
isParallel()	BaseStream	Returns true if the stream is parallel (that is, if the stream would execute in parallel when a terminal operation is executed)
sequential()	BaseStream	Returns a sequential stream
unordered()	BaseStream	Creates an unordered stream
forEachOrdered()	Stream	Consumes elements from a stream and performs an action on those elements, in the encounter order of the original stream if that stream has an order

Embarrassingly Parallel, Take Two (with Parallel Streams)

Earlier we talked about how some problems are “embarrassingly parallel”: that is, they’re easily split up into independent pieces that can be computed separately and then combined. These are precisely the kinds of problems that work well with parallel streams. The sum example we’ve been using in this section is a great example of an embarrassingly parallel problem: we can split up the stream into subsections, compute the sum of the subsection, and then combine the sums from each to make one total sum. We visualized this process in [Chapter 9](#) with two images. The first illustrates how we compute a sum with a sequential stream:

$$\begin{array}{cccccccccc}
 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
 \hline
 & 3 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
 \hline
 & 6 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
 \hline
 & 10 & 5 & 6 & 7 & 8 & 9 & 10 \\
 \hline
 & 15 & 6 & 7 & 8 & 9 & 10 \\
 \hline
 & 21 & 7 & 8 & 9 & 10 \\
 \hline
 & 28 & 8 & 9 & 10 \\
 \hline
 & 36 & 9 & 10 \\
 \hline
 & 45 & 10 \\
 \hline
 & & & & & 55
 \end{array}$$

while the second image illustrates how we can compute a sum with a parallel stream that is using two workers:



As you can see in the second illustration, the stream can be split in two and each sum computed completely independently from the other and then combined at the end. Again, this is precisely the kind of problem that works well with parallel streams.

We can summarize the conditions that make for successful parallel streams as follows: a problem is most suitable for parallel streams if the pipeline operations are *stateless*, the reduction operation used to compute the result is *associative* and *stateless*, and the stream is *unordered*.

Let's take a look at each of these.

Associative Operations

We talked about associative operations earlier in [Chapter 9](#). Recall from that chapter that `sum()` is an example of an associative operation. That is, we can compute the sum of $a + b$, and then add c , or we can compute $b + c$, and then add a , and get the same result.

Some operations are definitely not associative. We discovered in [Chapter 9](#) that computing the average of a stream of numbers is not associative. Recall that when we implemented our own average reduction operation, we got an incorrect result. Using the built-in `average()` method solved the problem in that chapter.

Because parallel streams are split up for processing in unpredictable ways, operations that aren't associative will probably fail. Just to reinforce how

important associativity is in properly reducing a stream, let's review how computing the average fails when we don't use the built-in `average()` method:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
OptionalDouble avg = nums
    .parallelStream()                      // make a parallel stream
    .mapToDouble(n -> n)                   // make a stream of doubles
    .reduce((d1, d2) -> (d1 + d2) / 2); // reduce with (bad) average
avg.ifPresent((a) ->
    System.out.println("Average of parallel stream with reduce: " + a));
OptionalDouble avg2 = nums
    .parallelStream()                      // make a parallel stream
    .mapToDouble(n -> n)                   // make a stream of doubles
    .average();                            // reduce with built-in average
avg2.ifPresent((a) ->
    System.out.println("Average of parallel stream with average: " + a));
```

In the first stream pipeline, we make the stream parallel, then we map to a stream of doubles so we can compute the average and return a double value; then we reduce using our own reduction function, which sums two numbers from the stream and divides by 2.

In the second stream pipeline, we do the same except we reduce with the built-in `average()` stream method, which is implemented in a way to produce the correct result (accounting for the inherent nonassociativity of the average function).

We get the output:

```
Average of parallel stream with reduce: 5.125 // wrong!
Average of parallel stream with average: 5.5   // correct!
```

The point is, just as with sequential stream reductions, parallel stream reductions need to be associative. This is even more important for parallel

streams given that we don't know how the system will split up and then recombine the stream results.

Stateless Operations (and Streams)

A stateless operation in a stream pipeline is an operation that does not depend on the context in which it's operating. Before we talk more about what stateless operations are, let's talk about what they are not.

There are two main ways you can create a *stateful* stream pipeline. The first is with *side effects*. We talked about this in [Chapter 9](#), too; side effects occur when your result creates or depends on changes to state in the pipeline.

Side Effects and Parallel Streams We already said not to ever modify the source of a stream from within the stream pipeline, so you haven't been—right? Right. That's one way you can create side effects, but you know not to do that.

Another way you can create side effects is to modify the field of an object. In sequential stream pipelines, you can get away with these kinds of side effects, and you may recall from [Chapter 9](#) how we created a list of DVDs from a file input stream. However, in parallel stream pipelines, these kinds of side effects will fail unless the objects you're modifying from within the pipeline are synchronized (either via a synchronized accessor method or if the objects are a concurrent data type).

Here's an example: We're streaming integers between 0 and 50 (noninclusive), filtering to find integers divisible by 10 and then summing those. We're also attempting to count the integers as we go by updating the field of an object. Although we aren't allowed to update a plain variable counter from within a lambda (you'll get a compiler error if you try to do this), we are allowed to update the field of an object, as we're doing here:

```

public static void stateful() {
    class Count {                                // an object to hold our counter
        int counter = 0;
    }
    Count count = new Count();                  // create an instance
    IntStream stream =                         // generate a stream of integers, 0-49
        IntStream.range(0, 50);
    int sum = stream
        .parallel()                            // make the stream parallel
        .filter(i -> {
            if (i % 10 == 0) {                // ...only count numbers divisible by 10
                count.counter++;             // ...there should be 5!
                return true;
            }
            return false;
        })
        .sum();                               // sum up the integers
    System.out.printf("sum: " + sum + ", count: " + count.counter);
}

```

The first time we ran this code, we got:

sum: 100, count: 5

The next time:

sum: 100, count: 4

Hmmm. We get the right sum (the sum doesn't depend on the counter and doesn't depend on the ordering), but the counter is not always correct (try a

bigger number than 50 in the range if this code doesn't produce different counter values for you). That's because, when we are using a parallel stream, multiple threads are accessing the `Count` object to modify the counter field and `Count` is not thread-safe.

We could use a synchronized object to store our counter and that would solve the problem, but it would also defeat the purpose of using parallel streams in this example.

We can fix the code above by removing the count in the parallel stream pipeline and computing it separately from the sum.

Stateful Operations in the Stream Pipeline Another way we can create stateful stream pipelines is with stateful stream operations. A stateful stream operation is one that requires some knowledge about the stream in order to operate. Take a look at the following stream pipeline:

```
IntStream stream = IntStream.range(0, 10);
long sum = stream.limit(5).sum();
System.out.println("Sum is: " + sum);
```

In this code, we are creating a stream of ten `ints`, limiting the stream to the first five of those `ints` (0–4) and summing those. The result, as you'd expect, is 10:

```
Sum is: 10
```

Now think about this operation, `limit(5)`. This is a stateful operation. Why? Because it requires context: the stream has to keep some intermediate state to know when it has five items and can stop streaming from the source (that is, short circuit the stream). So adding `parallel()` to this stream will not improve performance and might even hurt performance because now that state has to be synchronized across threads.

Let's test this and see what performance we get. We'll create a stream of 100 million `ints`, limit the stream to the first 5 `ints`, and sum, like we just did above. We'll also time the operation:

```
final int SIZE = 100_000_000;
final int LIMIT = 5;
long sum = 0, startTime, endTime, duration;
IntStream stream = IntStream.range(0, SIZE);
startTime = Instant.now().toEpochMilli();
sum = stream
    .limit(LIMIT)
    .sum();
endTime = Instant.now().toEpochMilli();
duration = endTime - startTime;
System.out.println("Items summed in " + duration
    + " milliseconds; sum is: " + sum);
```

When we run this on our machine (eight cores), we get

Items summed in 29 milliseconds; sum is: 10

Running it several times takes 28 or 29 milliseconds each time.

Now, let's make this a parallel stream and see what we get:

```
IntStream stream = IntStream.range(0, SIZE);
startTime = Instant.now().toEpochMilli();
sum = stream
    .parallel()
    .limit(LIMIT)
    .sum();
endTime = Instant.now().toEpochMilli();
duration = endTime - startTime;
System.out.println("Items summed in " + duration
+ " milliseconds; sum is: " + sum);
```

Running this, we get

```
Items summed in 34 milliseconds; sum is: 10
```

The performance is worse! Repeated runs yield running times of between 33 and 36 milliseconds each time. Increasing the `SIZE` to 400 million yields similar results.

One thing we should consider here is that the overhead of creating eight threads might be contributing to the performance problem. To really test that a parallel pipeline can hurt (or, at least, not help) when using `limit()`, we should try our experiment with a custom `ForkJoinPool` and set the number of threads ourselves. Let's do that.

The code is similar to what you've seen before: we create a custom `ForkJoinPool`, submit the task to the `ForkJoinPool`, get the result from the `FutureTask` that's returned, and time the whole thing. We've bumped up the size of the stream to 400 million ints, but we're still limiting to 5 ints for the sum. For this initial test, we've commented out the call to `parallel()` in the stream pipeline, so our first test will be on a sequential stream with one thread in the `ForkJoinPool`:

```
final int SIZE = 400_000_000;
final int LIMIT = 5;
long sum = 0, startTime, endTime, duration;
ForkJoinPool fjp = new ForkJoinPool(1);    // Limit FJP to 1 thread
IntStream stream = IntStream.range(0, SIZE);
try {
    startTime = Instant.now().toEpochMilli();
    sum =
        fjp.submit(
            () -> stream
                //.parallel()                      // test sequential first
                .limit(LIMIT)
                .sum()
        ).get();
    endTime = Instant.now().toEpochMilli();
    duration = endTime - startTime;
    System.out.println("FJP Stream data summed in "
        + duration + " milliseconds; sum is: " + sum);
} catch (Exception e) {
    System.out.println("Error executing stream sum");
    e.printStackTrace();
}
```

When we ran this code, we got

FJP Stream data summed in 35 milliseconds; sum is: 10

Now let's make the stream pipeline parallel and increase the number of

threads to two:

```
ForkJoinPool fjp = new ForkJoinPool(2); // Now use 2 threads
IntStream stream = IntStream.range(0, SIZE);
try {
    startTime = Instant.now().toEpochMilli();
    sum =
        fjp.submit(
            () -> stream
                .parallel()                      // make the stream parallel
                .limit(LIMIT)
                .sum()
        ).get();
    endTime = Instant.now().toEpochMilli();
    duration = endTime - startTime;
    System.out.println("FJP Stream data summed in "
        + duration + " milliseconds; sum is: " + sum);
} catch (Exception e) {
    System.out.println("Error executing stream sum");
    e.printStackTrace();
}
```

Run it again and we get

```
FJP Stream data summed in 36 milliseconds; sum is: 10
```

Now using parallel streams is not that much slower than running it sequential, but there's still no benefit to using parallel streams; our results from this test indicate that whether we use two threads or eight, the parallel pipeline runs no

faster than the sequential pipeline.

Run some more experiments yourself. Change the number of workers in the `ForkJoinPool`; see what happens. Change the `LIMIT` to a much larger number, like 500,000, and see what happens. More than likely, you'll find that the parallel stream pipeline runs the same as or more slowly than the sequential stream pipeline for this operation, which is what we'd expect.

We've talked about a couple of ways that stream pipelines are stateful: that is, stream pipelines are stateful if we create side effects (modifying an object as we process the pipeline) or if we use a stateful stream operation, like `limit()`. Other stateful stream operations might be fairly obvious to you; they are `skip()`, `distinct()`, and `sorted()`. How do you know which stream operations are stateful? The documentation describing these methods says so. And if you think about it, each of these operations requires some knowledge about the stream in order for the stream to operate.

Stream operations that are not necessarily stateful include `map()`, `filter()`, and `reduce()` (among others), so there's a lot we can do with streams that is stateless, although, as with all things related to concurrency, we need to be careful. As you saw in the example above, when we tried to count numbers in the stream using a `filter()`, we can make `filter()` stateful by creating a side effect.

Just to revisit quickly a simple example of a parallel stream using `map()`, `filter()`, and `reduce()` stateless operations, here's a slightly modified take on our original example:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
long sum = nums.stream()
    .parallel()                  // make the stream parallel
    .mapToInt(n -> n)           // map from Integer to int
    .filter(i ->                 // filter the evens
        i % 2 == 0 ? true : false)
    .sum();                      // sum the evens
System.out.println("Sum of evens is: " + sum);
```

Run this and you should see the following:

```
Sum of evens is: 30
```

None of these stream operations requires any state in order to operate on the values in the stream pipeline. The map simply converts the type of the stream value from `Integer` to `int`; the filter simply determines if the value is even or odd, and the reduction (`sum()`) sums the values, which can happen in any order because `sum()` is associative, so this operation requires no state either. This is an example of a stateless stream pipeline...well almost.

Hopefully now, you can see the difference between a stateless and a stateful stream pipeline. We've mostly talked about what a stateless stream pipeline is by talking about what a stateful stream pipeline is. The short story is that a stateless stream pipeline is one that requires no underlying intermediate state to be stored and accessed by the thread(s) in order to execute properly. And that statelessness is what helps make parallel stream pipelines more efficient.

However, we need to do one more thing to make this stream pipeline completely stateless, as you'll see next.

Unordered Streams

Earlier we said a problem is suitable for parallel streams if the operations used to compute the result are associative and stateless and the stream is unordered. We've talked about the first two; what about unordered?

By default, many (but not all!) streams are ordered. That means there is an inherent ordering to the items in the stream. A stream of `ints` created by `range()` and a stream of `Integers` created from a `List` of `Integers` are both ordered streams. Intuitively that makes sense; technically, ordering is determined by whether the stream has an `ORDERED` *characteristic*. This is just a bit that is set on the underlying implementation of a stream. There are other characteristics of streams, including `SORTED` and `DISTINCT`, but you don't need to worry too much about characteristics of streams except to know that the characteristics of a stream can affect how that stream performs, especially if you make the stream parallel.

Sometimes we want our streams to retain their order; for instance, if we are mapping stream values from `ints` to `Integer`, filtering to extract the even numbers, and then displaying the results, we might want the ordering of the stream to be maintained so we see the results in order.

However, an ordered stream pipeline will not execute as efficiently in parallel. Again, it comes down to context: an ordered stream has to maintain some state—in this case, the ordering of the stream values—to keep the stream

values in order, which adds overhead to the processing.



The characteristics of a stream can be inspected by using a `Spliterator`, which is an object for traversing a source, like a collection or a stream, that can also split up that source for potential parallel processing. `Spliterator` and its characteristics are not on the exam, but if you want to explore how streams (and collections) are traversed and partitioned in more detail, you can study `Spliterator` in depth.

Here's how you can use a stream's `spliterator()` method to determine if that stream is ordered:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
Stream<Integer> s = nums.stream();
System.out.println("Stream from List ordered? " +
    s.spliterator().hasCharacteristics(Spliterator.ORDERED));
```

In this case you'll see that, indeed, the stream of Integers we make from a List is ordered.

If a stream is ordered and we process it in parallel, the stream will remain ordered, but at the price of some efficiency. If we don't care about the ordering, as is the case when we are summing numbers, we might as well remove the ordering on the stream in scenarios where we are using a parallel stream pipeline. That way we get the extra performance benefits of working with an unordered stream in a situation where we shouldn't be concerned about the ordering anyway (because if we are, then the problem we're trying to solve probably isn't appropriate for a parallel stream pipeline).

So how do we make sure we're working with an unordered stream? We can explicitly tell the stream to not worry about remaining ordered by calling the `unordered()` stream method. Of course, we should do this *before* we call `parallel()` so we can maximize the efficiency of the parallel processing. Here's how we can modify the previous example to create an unordered stream

pipeline:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
long sum = nums.stream()
    .unordered()      // make the stream unordered
    .parallel()
    .mapToInt(n -> n)
    .filter(i -> i % 2 == 0 ? true : false)
    .sum();

System.out.println("Sum of evens is: " + sum);
```

Calling `unordered()` doesn't change the ordering of the stream; it just unsets that `ORDERED` bit so the stream doesn't have to maintain the ordering state.

Depending on how the stream is processed in the parallel pipeline, you may find that the final stream (before the reduction) is in the same order as the source, or not. Note that an ordered stream is not the same thing as a sorted stream. Once you've made a stream unordered, there is no way to order it again, except by calling `sorted()`, which makes it sorted *and* ordered but not necessarily in the same order as it was in the original source.

Don't worry too much about ordering for the exam; do remember that ordering has an implication for performance and make sure to note which collections create ordered streams (e.g., `List`) and which create unordered streams (e.g., `HashSet`) when you call that collection's `stream()` method, so you know the performance implications.

To summarize: to make the most efficient parallel stream, you should:

- Make sure your reductions are associative and stateless.
- Avoid side effects.
- Make sure your pipeline is stateless by avoiding contextual operations such as `limit()`, `skip()`, `distinct()`, and `sorted()`.

forEach() and forEachOrdered()

As you know, `forEach()` is a terminal stream operation that takes a `Consumer` and consumes each item in the stream. We often use `forEach()` to show the

values in a stream after a map and/or filter operation, for example:

```
dogs.stream().filter(d -> d.getAge() > 7).foreach(System.out::println);
```

will display any Dog in the stream whose age is > 7 in the order in which the values in the stream are encountered.

Imagine we have a Dog class and a Dog constructor that takes the name and age of a dog. You've seen the Dog class often enough that we probably don't have to repeat it. Given that Dog class, we can make some dogs, like this:

```
List<Dog> dogs = new ArrayList<>();
Dog aiko = new Dog("aiko", 10);
Dog boi = new Dog("boi", 6);
Dog charis = new Dog("charis", 7);
Dog clover = new Dog("clover", 12);
Dog zooey = new Dog("zooey", 8);
dogs.add(aiko); dogs.add(boi); dogs.add(charis);
dogs.add(clover); dogs.add(zooey);
```

Notice that we've added the dogs in alphabetical order on purpose, so we can watch the ordering of our stream when we use a sequential stream and when we use a parallel stream.

Running the code:

```
dogs.stream().filter(d -> d.getAge() > 7).foreach(System.out::println);
```

gives us the output:

```
aiko is 10 years old
clover is 12 years old
zooey is 8 years old
```

That is, we see all dogs whose age is > 7 in the order in which they are encountered in the stream (which is the order in which they are added to the List). This is because, by default, when we stream the List, we get a sequential,

ordered stream.

Now, as you might expect at this point, when we make this a parallel stream, we get different results:

```
dogs.stream()
    .parallel()
    .filter(d -> d.getAge() > 7)
    .foreach(System.out::println);
```

This code produced the output:

```
zooey is 8 years old
aiko is 10 years old
clover is 12 years old
```

The output is unpredictable; if you run it again, you may get different results. Even though the stream is ordered (since we're streaming a `List`), we see the output in a random order because the stream is parallel, so the dogs can be processed by different threads that may finish at different times, and the `foreach()` consumer is stateful: it is essentially creating a side effect by outputting data to the console, which is not thread-safe.

We can make sure we see the dogs in the order in which they appear in the original source by using the method `foreachOrdered()` instead:

```
dogs.stream().parallel().filter(d -> d.getAge() > 7)
    .foreachOrdered(System.out::println); // enforce ordering
```

We can do this because the underlying stream is ordered (it's made from a `List` and its `ORDERED` characteristic is set), and the `foreachOrdered()` method will make sure the items are seen in the same order as they are encountered in the ordered stream. (Note that if we call `unordered()` on the stream, then the order will not be guaranteed!)

Of course, this takes a bit of overhead to perform, right? As we discussed earlier, an ordered stream is going to be less efficient when processed in parallel than an unordered stream. However, there may be times when you want to maintain the ordering of the stream when processing in parallel and you're

willing to sacrifice a bit of efficiency.

Typically, you won't end up using `forEachOrdered()` much in the real world; you're sacrificing performance, and usually you don't want to see the results of a big data operation on a parallel stream; you want to collect the results in a new data structure or compute some final result like a sum. However, you will see `forEachOrdered()` on the exam, so make sure you understand how it works.

Also, note that `forEach()` (and `forEachOrdered()`) as well as `peek()` are stream operations that are designed for side effects. That is, they consume elements from the stream: `peek()` typically creates a side effect (say, printing the stream element passing through) and then passes the values on unchanged to the stream; `forEach()` consumes the stream values and produces a result (it's a terminal operation), and typically that result is output to the console or saves each item to the field of an object. So take care when using `forEach()`, `forEachOrdered()`, and `peek()` with parallel streams, remembering that stream pipelines with side effects (even just printing to the console) can change how the stream pipeline operates, and reduce the performance of parallel streams.

A Quick Word About `findAny()`

You might remember from [Chapter 9](#) that we used the `findAny()` stream operation to find any value in the stream pipeline that matched a `filter()`. For instance, we can use `findAny()` to find any even `int` in a stream like this:

```
IntStream nums = IntStream.range(0, 20);
OptionalInt any = nums
    .filter(i -> // filter the evens
        i % 2 == 0 ? true : false)
    .findAny(); // find any even int
any.ifPresent(i -> System.out.println("Any even is: " + i));
```

With a sequential stream, `findAny()` will likely return the first value in the stream, 0, every time, even though it's not guaranteed.

Now, let's parallelize this stream and see what happens. We'll add a `peek()`, so we can see the thread workers as they work on the problem:

```
IntStream nums = IntStream.range(0, 20);
OptionalInt any = nums
    .parallel()           // make the stream parallel
                        // peek at the thread name
    .peek(i -> System.out.println(i + ": "
        + Thread.currentThread().getName()))
    .filter(i ->      // filter the evens
        i % 2 == 0 ? true : false)
    .findAny();           // find any even int
any.ifPresent(i -> System.out.println("Any even is: " + i));
```

We run this and get

```
12: main
1: ForkJoinPool.commonPool-worker-6
5: ForkJoinPool.commonPool-worker-7
2: ForkJoinPool.commonPool-worker-3
6: ForkJoinPool.commonPool-worker-1
8: ForkJoinPool.commonPool-worker-5
17: ForkJoinPool.commonPool-worker-2
16: ForkJoinPool.commonPool-worker-4
Any even is: 12
```

Remember, our computer has eight cores, so this stream pipeline has been split up into eight workers, each tackling part of the stream. The `findAny()` method is short circuiting, so even though we still have 12 more values in the stream to process (since the stream has 20 values), we stop as soon as we find the first even number. In this run, we had several threads with even numbers: the main worker, worker-1, worker-4, and worker-5. It just so happens that the main thread probably got done first, and so as soon as that even number was found,

everything else stopped, and that result, 12, was returned.

Run the code again, and you'll likely get a different answer.

This example illustrates that `findAny()` really does find any result, particularly when you're working with a parallel stream pipeline.

A Parallel Stream Implementation of a RecursiveTask

Let's write code to sum an array of `ints` with a `ForkJoinPool` `RecursiveTask`, and compare that code with a parallel stream (which, remember, uses `ForkJoinPool` under the covers). We'll also implement it with a plain-old `for` loop, so you can see not only how the code itself compares, but also how the performance compares. Much of this code should look familiar to you by now.

```

/*
 * Sum numbers in an array of SIZE random numbers
 * from 1 to MAX if number > NUM
 */
public class SumRecursiveTask extends RecursiveTask<Long> {
    public static final int SIZE = 400_000_000;
    public static final int THRESHOLD = 1000;
    public static final int MAX = 10;          // array of numbers, 1-10
    public static final int NUM = 5;           // sum numbers > 5
    private int[] data;
    private int start;
    private int end;
    public SumRecursiveTask(int[] data, int start, int end) {
        this.data = data;
        this.start = start;
        this.end = end;
    }
    @Override
    protected Long compute() {
        long tempSum = 0;
        if (end - start <= THRESHOLD) {
            for (int i = start; i < end; i++) {
                if (data[i] > NUM) {
                    vtempSum += data[i];
                }
            }
            return tempSum;
        } else {
            int halfWay = ((end - start) / 2) + start;
            SumRecursiveTask t1 = new SumRecursiveTask(data, start, halfWay);
            SumRecursiveTask t2 = new SumRecursiveTask(data, halfWay, end);
            t1.fork();           // queue left half of task
            long sum2 = t2.compute(); // compute right half
            long sum1 = t1.join();   // compute left and join
            return sum1 + sum2;
        }
    }
    public static void main(String[] args) {
        int[] data2sum = new int[SIZE];
        long sum = 0, startTime, endTime, duration;
        // create an array of random numbers between 1 and MAX
        for (int i = 0; i < SIZE; i++) {
            data2sum[i] = ThreadLocalRandom.current().nextInt(MAX) + 1;
        }
        startTime = Instant.now().toEpochMilli();
        // sum numbers with plain old for loop
        for (int i = 0; i < data2sum.length; i++) {
            if (data2sum[i] > NUM) {

```



```

        sum = sum + data2sum[i];
    }
}
endTime = Instant.now().toEpochMilli();
duration = endTime - startTime;
System.out.println("Summed with for loop in " + duration
+ " milliseconds; sum is: " + sum);

// sum numbers with RecursiveTask
ForkJoinPool fjp = new ForkJoinPool();
SumRecursiveTask action =
    new SumRecursiveTask(data2sum, 0, data2sum.length);
startTime = Instant.now().toEpochMilli();
sum = fjp.invoke(action);
endTime = Instant.now().toEpochMilli();
duration = endTime - startTime;
System.out.println("Summed with recursive task in "
+ duration + " milliseconds; sum is: " + sum);

// sum numbers with a parallel stream
IntStream stream2sum = IntStream.of(data2sum);
startTime = Instant.now().toEpochMilli();
sum =
    stream2sum
        .unordered()
        .parallel()
        .filter(i -> i > NUM)
        .sum();
endTime = Instant.now().toEpochMilli();
duration = endTime - startTime;
System.out.println("Stream data summed in " + duration
+ " milliseconds; sum is: " + sum);

// sum numbers with a parallel stream, limiting workers
ForkJoinPool fjp2 = new ForkJoinPool(4);
IntStream stream2sum2 = IntStream.of(data2sum);
try {
    startTime = Instant.now().toEpochMilli();
    sum =
        fjp2.submit(
            () -> stream2sum2
                .unordered()
                .parallel()
                .filter(i -> i > NUM)
                .sum()
        ).get();
    endTime = Instant.now().toEpochMilli();
    duration = endTime - startTime;
    System.out.println("FJP4 Stream data summed in "
+ duration + " milliseconds; sum is: " + sum);
} catch (Exception e) {
}

```

```
        System.out.println("Error executing stream average");
        e.printStackTrace();
    }
}
}
```

This code first generates a large array of random numbers between 1 and 10 and then computes the sum of all numbers > 5 four different times (from the same data, so we should get the same answer each time): first, using a plain-old `for` loop; second, using a `RecursiveTask`, third, using a parallel stream on the default `ForkJoinPool` (that is, using all eight cores of our machine); and finally, using a parallel stream on a custom `ForkJoinPool` with four workers (using four cores).

Here are our results:

```
Summed with for loop in 287 milliseconds; sum is: 399980957
Summed with recursive task in 267 milliseconds; sum is: 399980957
Stream data summed in 118 milliseconds; sum is: 399980957
FJP4 Stream data summed in 136 milliseconds; sum is: 399980957
```

We ran it again and this time we got

```
Summed with for loop in 292 milliseconds; sum is: 399927370
Summed with recursive task in 196 milliseconds; sum is: 399927370
Stream data summed in 184 milliseconds; sum is: 399927370
FJP4 Stream data summed in 138 milliseconds; sum is: 399927370
```

In the first run, the parallel stream running on all eight cores was the winner, slightly faster than the parallel stream running on four cores. Both parallel streams were far superior to the `RecursiveTask` and the `for` loop.

In the second run, this time the parallel stream running on four cores was the clear winner, with the `RecursiveTask` and the parallel stream running on eight cores about the same, both much faster than the `for` loop.

As you can see, your results will vary depending on the solution you choose as well as your underlying machine architecture. If you get an `OutOfMemoryError` when you run this, try reducing the `SIZE`.

Reducing Parallel Streams with `reduce()`

Let's say you want to reduce your stream to a value, but none of the built-in stream reduction methods (like `sum()`) suffice. You can build your own custom reduction with the `reduce()` function, as you saw in [Chapter 9](#).

How about building a custom reduction that multiplies all the elements of a stream? It's a bit like computing the sum, except we're multiplying instead. This is known as computing the *product* (as compared to the sum). Before we write this reduction, we should check a few things. First, remember that a reduction produces an `Optional` value unless we provide an identity.

Recall the type signature of `reduce()`:

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

Although the identity is optional, providing one is a good idea because it allows us to get a result (rather than an `Optional` result), and that identity value is also used to make the first result in the stream pipeline, and we want to make sure the correct identity is being used in our custom reduction.

For the sum, the identity value is 0 because any number added to 0 is that number. For the product, the identity is 1 because any number multiplied by 1 is that number. So, we'll use 1. That's easy enough.

The accumulator is a `BinaryOperator` that takes two values and produces one value. Remember that an `Operator` produces a value of the same type as its arguments. Our `BinaryOperator` is simple; it just takes two numbers and multiples them together, returning a number:

```
(i1, i2) -> i1 * i2
```

Another thing we should check is that our accumulator function is associative. Remember the trouble we ran into when we tried to reduce using a custom average function? We didn't get a correct result because our average function is not associative, so we had to use the built-in `average()` reduction method instead.

Is our product function associative? It is. That is, we can multiply $a * b$ and then by c and get the same answer as when we multiply $b * c$ and then by a .

Another quick check: Is our accumulator stateless? That is, does it rely on any additional state in the stream to be computed? If it's not stateless, then we could run into trouble; either we'll potentially get incorrect results or we'll drastically reduce the performance of the parallel stream (or both!). In this case, our reduction function is, indeed, stateless. There is no state needed in order to properly multiply two values from the stream and produce a result.

Okay! We've got our identity, and we've got our associative, stateless reduction function. Now we can write the code to reduce a parallel stream to the product of all values in the stream:

```
List<Integer> nums = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
int mult = nums.stream()
    .unordered()                      // unordered for efficiency
    .parallel()                        // make the stream parallel
    .reduce(1, (i1, i2) -> i1 * i2);  // reduce to the product
System.out.println("Product reduction: " + mult);
```

Here, we start with a `List` of `Integer` values and create a stream. We call `unordered()` on that stream so we can get additional efficiency from parallel processing—and this works great, because just like addition, computing the product does not depend on the order of the values. We then make the stream parallel and call the `reduce()` terminal operation method, passing in our identity, 1, and our product accumulator. Here is the result we get:

Product reduction: 3628800

That's the correct answer (phew!). Running the code again several times produces the same result each time. It looks like we got the code correct.

As an exercise, you can try this reduction on a much larger stream of numbers and time the performance, comparing the parallel version with the sequential version. See what results you get!

Collecting Values from a Parallel Stream

As we've said before, when processing data with streams, you will most likely want to reduce the stream to a single value, like a sum or a product or a count, or you might want to collect results into a new collection with `collect()` (which is

a reduction, too).

Consider again our `ArrayList` of dogs we used above, containing aiko, boi, charis, clover, and zooey.

If you forget that you aren't supposed to create side effects from a parallel stream pipeline, you might do something like this to collect dogs who are older than 7 into a new `List` collection:

```
List<Dog> dogsOlderThan7 = new ArrayList<>();  
long count = dogs.stream() // stream the dogs  
    .unordered() // make the stream unordered  
    .parallel() // make the stream parallel  
    .filter(d -> d.getAge() > 7) // filter the dogs  
    .peek(d -> dogsOlderThan7.add(d)) // save... with a side effect  
    .count();  
  
System.out.println("Dogs older than 7, via side effect: " + dogsOlderThan7);
```

Here, we're initializing a new `List`, `dogsOlderThan7`; we're streaming our original dogs `ArrayList`, making it unordered, making it parallel, filtering for dogs older than 7, and then making the mistake of saving those dogs older than 7 to our new `List` using a side effect from within a `peek()` lambda expression. We then terminate the pipeline with the `count()` operation.

The first time we ran this we got the following:

```
Dogs older than 7, via side effect: [null, aiko is 10 years old]
```

Hmm, definitely a problem there! Running it again, we got this:

```
Dogs older than 7, via side effect: [null, null, clover is 12 years old]
```

Clearly, as you should know well by now, this is not the way to collect results from a stream pipeline. The reason this fails is because we have multiple threads trying to access the `List` `dogsOlderThan7` at the same time and `dogsOlderThan7` is not thread-safe.

We could fix this code by using a synchronized list, like this:

```
List<Dog> dogsOlderThan7 =  
    Collections.synchronizedList(new ArrayList<>());
```

Now, the `List` is thread-safe. The order of the dogs in the resulting `List` may not be the same as in the source, but at least you'll get the right set of dogs. However, you're also sacrificing some performance because you're forcing a synchronization of all the threads attempting to write to the synchronized `List`.

A better way to collect values from a parallel stream pipeline is to use `Collectors`, as we did in [Chapter 9](#) with sequential streams. Here's how:

```
List<Dog> dogsOlderThan7 =  
    dogs.stream() // stream the dogs  
        .unordered() // make the stream unordered  
        .parallel() // make the stream parallel  
        .filter(d -> d.getAge() > 7) // filter dogs older than 7  
        .collect(Collectors.toList()); // collect older dogs into a List  
System.out.println("Dogs older than 7: " + dogsOlderThan7);
```

When we run this code, we see the output:

```
Dogs older than 7: [aiko is 10 years old, clover is 12 years old,  
zooey is 8 years old]
```

Now you might be thinking, “Didn't you just say that the `List` is not thread-safe? We're still using a `List` in the `collect()` method with `Collectors.toList()`, so how is this working?”

Good question. It turns out that the way `collect()` works under the covers is that each worker processing a piece of the stream collects its data into its own collection. Worker 1 will create a `List` of dogs older than 7; worker 2 will create a `List` of dogs older than 7; and so on. Each of these `Lists` is separate, built from the pieces of the stream that each worker got when the stream was split up across the parallel workers. At the end, once each thread is complete, the separate `Lists` are merged together, in a thread-safe manner, to create one final `List` of dogs.

`Lists` are inherently ordered, so we will get a reduction in efficiency unless

we make sure the stream is unordered, which we've done by adding a call to `unordered()` at the beginning of the pipeline. (Of course, this means the dogs may not be in the same order as they were in the source, but you expected that, right?). You can also collect values from a parallel stream with `toSet()` and `toMap()`, and a method `toConcurrentMap()` is there for additional efficiency since merging maps is expensive.

As we mentioned in [Chapter 9](#), you can use the `collect()` method with your own supplier, accumulator, and combiner functions if you need a way to collect that isn't provided for by the `Collectors` class:

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

If you use a custom collector, just like with the custom reduction, you'll need to make sure that your accumulator is associative and stateless. In addition, your combiner should be stateless and thread-safe. Typically, the combiner will be adding values to a new collection, so using a concurrent collection here is a good idea.

CERTIFICATION SUMMARY

This chapter covered the required concurrency knowledge you'll need to apply on the certification exam. The `java.util.concurrent` package and its subpackages form a high-level multithreading framework in Java. You should become familiar with threading basics before attempting to apply the Java concurrency libraries, but once you learn `java.util.concurrent`, you may never extend `Thread` again.

`Callables` and `Executors` (and their underlying thread pools) form the basis of a high-level alternative to creating new `Threads` directly. As the trend of adding more CPU cores continues, knowing how to get Java to make use of them all concurrently could put you on easy street. The high-level APIs provided by `java.util.concurrent` help you create efficient multithreaded applications while eliminating the need to use low-level threading APIs such as `wait()`, `notify()`, and `synchronized`, which can be a source of hard-to-detect bugs.

When using an `Executor`, you will commonly create a `Callable` implementation to represent the work that needs to be executed concurrently. A

`Runnable` can be used for the same purpose, but a `Callable` leverages generics to allow a generic return type from its `call` method. `Executor` or `ExecutorService` instances with predefined behavior can be obtained by calling one of the factory methods in the `Executors` class like so: `ExecutorService es = Executors.newFixedThreadPool(100);`

Once you obtain an `ExecutorService`, you submit a task in the form of a `Runnable` or `Callable`, or a collection of `Callable` instances to the `ExecutorService` using one of the `execute`, `submit`, `invokeAny`, or `invokeAll` methods. An `ExecutorService` can be held onto during the entire life of your application if needed, but once it is no longer needed, it should be terminated using the `shutdown` and `shutdownNow` methods.

We looked at the Fork/Join Framework, which supplies a highly specialized type of `Executor`. Use the Fork/Join Framework when the work you would typically put in a `Callable` can be split into multiple units of work. The purpose of the Fork/Join Framework is to decrease the amount of time it takes to solve a problem by leveraging the additional CPUs in a system. You should only run a single Fork/Join task at a time in an application, because the goal of the framework is to allow a single task to consume all available CPU resources in order to be solved as quickly as possible. In most cases, the effort of splitting a single task into multiple tasks that can be operated on by the underlying Fork/Join threads will introduce additional overhead. Don't assume that applying Fork/Join will grant you a performance benefit for all problems. The overhead involved may be large enough that any benefit of applying the framework is offset.

When applying the Fork/Join Framework, first subclass either `RecursiveTask` (if a return result is desired) or `RecursiveAction`. Within one of these `ForkJoinTask` subclasses, you must implement the `compute` method. The `compute()` method is where you divide the work of a task into parts and then call the `fork` and `join` methods or the `invokeAll` method. To execute the task, create a `ForkJoinPool` instance with `ForkJoinPool pool = new ForkJoinPool();` and submit the `RecursiveTask` or `RecursiveAction` to the pool with the `pool.invoke(task)` method. Although the Fork/Join API itself is not that large, creating a correct and efficient implementation of a `ForkJoinTask` can be challenging.

Java 8 added parallel streams to make the Fork/Join Framework easier to use. Parallel streams are built on top of the Fork/Join Framework and allow you to split a task into parts more easily than when using the Fork/Join API directly. Simply create a stream and call `parallel()`, and your stream pipeline will be

split into tasks that execute in separate threads. However, as you saw, you need to be aware of the potential issues with parallel streams and make sure you are using them to solve the appropriate kinds of problems.

We learned about the `java.util.concurrent` collections. There are three categories of collections: copy-on-write collections, concurrent collections, and blocking queues. The copy-on-write and concurrent collections are similar in use to the traditional `java.util` collections but are designed to be used efficiently in a thread-safe fashion. The copy-on-write collections (`CopyOnWriteArrayList` and `CopyOnwriteArraySet`) should be used for read-heavy scenarios. When attempting to loop through all the elements in one of the copy-on-write collections, always use an `Iterator`. The concurrent collections included

- `ConcurrentHashMap`
- `ConcurrentLinkedDeque`
- `ConcurrentLinkedQueue`
- `ConcurrentSkipListMap`
- `ConcurrentSkipListSet`

These collections are meant to be used concurrently without requiring locking. Remember that iterators of these five concurrent collections are weakly consistent. `ConcurrentHashMap` and `ConcurrentSkipListMap` are `ConcurrentMap` implementations that add atomic `putIfAbsent`, `remove`, and `replace` methods to the `Map` interface. Seven blocking queue implementations are provided by the `java.util.concurrent` package:

- `ArrayBlockingQueue`
- `LinkedBlockingDeque`
- `LinkedBlockingQueue`
- `PriorityBlockingQueue`
- `DelayQueue`
- `LinkedTransferQueue`
- `SynchronousQueue`

These blocking queues are used to exchange objects between threads—one thread will deposit an object and another thread will retrieve that object. Depending on which queue type is used, the parameters used to create the queue,

and the method being called, an insert or a removal operation may block until it can be completed successfully. In Java 7, the `LinkedTransferQueue` class was added and acts as a superset of several blocking queue types; you should prefer it when possible.

Another way to coordinate threads is to use a `CyclicBarrier`. A `CyclicBarrier` creates a barrier where all threads must wait until all participating threads reach that barrier; once they do, then the threads can continue. You can have an optional `Runnable` run before the threads continue; the last thread to reach the barrier is the thread that's used to run that `Runnable`.

The `java.util.concurrent.atomic` and `java.util.concurrent.locks` packages contain additional utility classes you might consider using in concurrent applications. The `java.util.concurrent.atomic` package supplies thread-safe classes that are similar to the traditional wrapper classes (such as `java.lang.Integer`) but with methods that support atomic modifications. The `java.util.concurrent.locks.Lock` interface and supporting classes enable you to create highly customized locking behaviors that are more flexible than traditional object monitor locking (the `synchronized` keyword).



TWO-MINUTE DRILL

Here are some of the key points from the certification objectives in this chapter.

Apply Atomic Variables and Locks (OCP Objective 10.3)

- ❑ The `java.util.concurrent.atomic` package provides classes that are similar to volatile fields (changes to an atomic object's value will be correctly read by other threads without the need for synchronized code blocks in your code).
- ❑ The atomic classes provide a `compareAndSet` method that is used to validate that an atomic variable's value will only be changed if it matches an expected value.
- ❑ The atomic classes provide several convenience methods such as `addAndGet` that will loop repeatedly until a `compareAndSet` succeeds.
- ❑ The `java.util.concurrent.locks` package contains a locking

mechanism that is an alternative to synchronized methods and blocks. You get greater flexibility at the cost of a more verbose syntax (such as having to manually call `lock.unlock()` and having an automatic release of a synchronization monitor at the end of a synchronized code block).

- ❑ The `ReentrantLock` class provides the basic Lock implementation. Commonly used methods are `lock()`, `unlock()`, `isLocked()`, and `tryLock()`. Calling `lock()` increments a counter and `unlock()` decrements the counter. A thread can only obtain the lock when the counter is zero.
- ❑ The `ReentrantReadWriteLock` class provides a `ReadWriteLock` implementation that supports a read lock (obtained by calling `readLock()`) and a write lock (obtained by calling `writeLock()`).

Use `java.util.concurrent` Collections (OCP Objective 10.4)

- ❑ Copy-on-write collections work well when there are more reads than writes because they make a new copy of the collection for each write. When looping through a copy-on-write collection, use an iterator (remember, `for-each` loops use an iterator).
- ❑ None of the concurrent collections make the elements stored in the collection thread-safe—just the collection itself.
- ❑ `ConcurrentHashMap`, `ConcurrentSkipListMap`, and `ConcurrentSkipListSet` should be preferred over synchronizing with the more traditional collections.
- ❑ `ConcurrentHashMap` and `ConcurrentSkipListMap` are `ConcurrentMap` implementations that enhance a standard Map by adding atomic operations that validate the presence and value of an element before performing an operation: `putIfAbsent(K key, V value)`, `remove(Object key, Object value)`, `replace(K key, V value)`, and `replace(K key, V oldValue, V newValue)`.
- ❑ Blocking queues are used to exchange objects between threads. Blocking queues will block (hence the name) when you call certain operations, such as calling `take()` when there are no elements to take. There are seven different blocking queues that have slightly different behaviors; you should be able to identify the behavior of each type.

Blocking Queue	Description
ArrayBlockingQueue	A FIFO (first-in-first-out) queue in which the head of the queue is the oldest element and the tail is the newest. An int parameter to the constructor limits the size of the queue (it is a bounded queue).
LinkedBlockingDeque	Similar to LinkedBlockingQueue, except it is a double-ended queue (deque). Instead of only supporting FIFO operations, you can remove from the head or tail of the queue.
LinkedBlockingQueue	A FIFO queue in which the head of the queue is the oldest element and the tail is the newest. An optional int parameter to the constructor limits the size of the queue (it can be bounded or unbounded).
PriorityBlockingQueue	An unbounded queue that orders elements using Comparable or Comparator. The head of the queue is the lowest value.
DelayQueue	An unbounded queue of java.util.concurrent.Delayed instances. Objects can only be taken once their delay has expired. The head of the queue is the object that expired first.
LinkedTransferQueue	Added in Java 7. An unbounded FIFO queue that supports the features of a ConcurrentLinkedQueue, SynchronousQueue, and LinkedBlockingQueue.
SynchronousQueue	A blocking queue with no capacity. An insert operation blocks until another thread executes a remove operation. A remove operation blocks until another thread executes an insert operation.

- ❑ Some blocking queues are bounded, meaning they have an upper bound on the number of elements that can be added, and a thread calling `put(e)` may block until space becomes available.
- ❑ `CyclicBarrier` creates a barrier at which threads must wait until all participating threads reach that barrier. Once all of the threads have reached the barrier, they can continue running. You can use `CyclicBarrier` to coordinate threads so that an action occurs only after another action is complete or to manage data in Collections that are not thread-safe.
- ❑ `CyclicBarrier` takes the number of threads that can wait at the barrier and an optional `Runnable` that is run after all threads reach the barrier, but before they continue execution. The last thread to reach the barrier is used to run this `Runnable`.

Use Executors and ThreadPools (OCP Objective 10.1)

- ❑ An `Executor` is used to submit a task for execution without being coupled to how or when the task is executed. Basically, it creates an abstraction that can be used in place of explicit thread creation and execution.
- ❑ An `ExecutorService` is an enhanced `Executor` that provides additional functionality, such as the ability to execute a `Callable` instance and to shut down (nondaemon threads in an `Executor` may keep the JVM running after your main method returns).
- ❑ The `Callable` interface is similar to the `Runnable` interface, but adds the ability to return a result from its `call` method and can optionally throw an exception.
- ❑ The `Executors` (plural) class provides factory methods that can be used to construct `ExecutorService` instances, for example: `ExecutorService ex = Executors.newFixedThreadPool(4);`.

Use the Parallel Fork/Join Framework (OCP Objective 10.5)

- ❑ Fork/Join enables work stealing among worker threads in order to keep all CPUs utilized and to increase the performance of highly parallelizable

tasks.

- A pool of worker threads of type `ForkJoinWorkerThread` is created when you create a new `ForkJoinPool()`. By default, one thread per CPU is created.
- To minimize the overhead of creating new threads, you should create a single Fork/Join pool in an application and reuse it for all recursive tasks.
- A Fork/Join task represents a large problem to solve (often involving a collection or array).
- When executed by a `ForkJoinPool`, the Fork/Join task will subdivide itself into Fork/Join tasks that represent smaller segments of the problem to be solved.
- A Fork/Join task is a subclass of the `ForkJoinTask` class, either `RecursiveAction` or `RecursiveTask`.
- Extend `RecursiveTask` when the `compute()` method must return a value, and extend `RecursiveAction` when the return type is `void`.
- When writing a `ForkJoinTask` implementation's `compute()` method, always call `fork()` before `join()` or use one of the `invokeAll()` methods instead of calling `fork()` and `join()`.
- You do not need to shut down a Fork/Join pool before exiting your application because the threads in a Fork/Join pool typically operate in daemon mode.

Use Parallel Streams Including Reduction, Decomposition, Merging Processes, Pipelines, and Performance (OCP Objective 10.6)

- Parallel streams are built on top of the Fork/Join pool.
- Parallel streams provide an easier syntax for creating tasks in the Fork/Join pool.
- You can use the default Fork/Join pool, or create a custom pool and submit tasks expressed as parallel streams via a `Callable`.
- Parallel streams split the stream into subtasks that represent portions of the problem to be solved. Each subtask solution is then combined to produce a final result for the terminal operation of the parallel stream.
- Create a parallel stream by calling `parallel()` on a stream object or

`parallelStream()` on a Collection object.

- Make a parallel stream sequential again by calling the `sequential()` method.
- Test to see if a stream is parallel with the `isParallel()` method.
- Parallel stream pipelines should be stateless, and for optimum performance, parallel streams should be unordered. Reduction operations on parallel streams should be associative and stateless.
- Stateful parallel stream pipelines will either create an error or unexpected results.
- Nonassociative reductions on streams will produce unexpected results.
- Just like sequential streams, parallel streams can have multiple intermediate operations and must have one terminal operation to produce a result.
- Test your parallel streams to verify you are getting the performance benefits you expect. The performance overhead of creating threads can be greater than the performance gain of a parallel stream in some situations.
- Stateful stream operations, such as `distinct()`, `limit()`, `skip()`, and `sorted()`, will limit the performance of your parallel streams.
- Collect results from a parallel stream pipeline with `collect()`, just as we did with sequential streams. The collecting happens in a thread-safe way, so you can use `collect()` with `Collectors.toList()`, `toSet()`, and `toMap()` safely.

Q SELF TEST

The following questions might be some of the hardest in the book. It's just a difficult topic, so don't panic. (We know some Java book authors who didn't do well with these topics and still managed to pass the exam.)

1. The following block of code creates a `CopyOnWriteArrayList`, adds elements to it, and prints the contents:

```
CopyOnWriteArrayList<Integer> cowList = new CopyOnWriteArrayList<>();
cowList.add(4);
cowList.add(2);
Iterator<Integer> it = cowList.iterator();
cowList.add(6);
while(it.hasNext()) {
    System.out.print(it.next() + " ");
}
```

What is the result?

- A. 6
 - B. 12
 - C. 4 2
 - D. 4 2 6
 - E. Compilation fails
 - F. An exception is thrown at runtime
2. Given:

```
CopyOnWriteArrayList<Integer> cowList = new CopyOnWriteArrayList<>();
cowList.add(4);
cowList.add(2);
cowList.add(6);
Iterator<Integer> it = cowList.iterator();
cowList.remove(2);
while(it.hasNext()) {
    System.out.print(it.next() + " ");
}
```

Which shows the output that will be produced?

- A. 12
 - B. 10
 - C. 4 2 6
 - D. 4 6
 - E. Compilation fails
 - F. An exception is thrown at runtime
3. Which methods from a `CopyOnWriteArrayList` will cause a new copy of the internal array to be created? (Choose all that apply.)
- A. add
 - B. get
 - C. iterator
 - D. remove
4. Given:
- ```
ArrayBlockingQueue<Integer> abq = new ArrayBlockingQueue<>(10);
```
- Which operation(s) can block indefinitely? (Choose all that apply.)
- A. `abq.add(1);`
  - B. `abq.offer(1);`
  - C. `abq.put(1);`
  - D. `abq.offer(1, 5, TimeUnit.SECONDS);`
5. Given the following code fragment:

```
class SingletonTestDrive {
 public static void main(String args[]) {
 CyclicBarrier barrier = new CyclicBarrier(3, () -> {
 System.out.println(Singleton.INSTANCE.getValue());
 });
 Runnable r = () -> {
 for (int i = 0; i < 100; i++) {
 Singleton.INSTANCE.updateValue();
 }
 try {
 barrier.await();
 } catch (InterruptedException | BrokenBarrierException e) {
 e.printStackTrace();
 }
 };
 Thread t1 = new Thread(r);
 Thread t2 = new Thread(r);
 Thread t3 = new Thread(r);
 t1.start(); t2.start(); t3.start();
 System.out.println("Main thread is complete");
 }
}
enum Singleton {
 INSTANCE;
 int value = 0;
 private void doSomethingWithValue() {
 value = value + 1;
 }
}
```

```
public synchronized int updateValue() {
 doSomethingWithValue();
 return value;
}
public int getValue() {
 return value;
}
}
```

What do you expect the output to be, and which thread(s) will be responsible for the output? (Choose all that apply.)

A.

Main thread is complete

300

Main thread, thread t3

B.

300

Main thread is complete

thread t3, Main thread

C.

Main thread is complete

300

OR

300

Main thread is complete

Main thread, the last thread to reach the barrier

D.

The total value displayed could be any number because the Singleton is not thread-

Main thread is complete

Main thread, the last thread to reach the barrier

E.

Main thread is complete

A BrokenBarrierException

Main thread, thread t1

**6.** Given:

```
ConcurrentMap<String, Integer> ages = new ConcurrentHashMap<>();
ages.put("John", 23);
```

Which method(s) would delete John from the map only if his value was still equal to 23?

- A. ages.delete("John", 23);
  - B. ages.deleteIfEquals("John", 23);
  - C. ages.remove("John", 23);
  - D. ages.removeIfEquals("John", 23);
- 7.** Which method represents the best approach to generating a random number between 1 and 10 if the method will be called concurrently and repeatedly by multiple threads?

A. public static int randomA() {  
    Random r = new Random();  
    return r.nextInt(10) + 1;  
}

B. private static Random sr = new Random();  
    public static int randomB() {  
        return sr.nextInt(10) + 1;  
    }

C. public static int randomC() {  
    int i = (int)(Math.random() \* 10 + 1);  
    return i;  
}

D. public static int randomD() {  
    ThreadLocalRandom lr = ThreadLocalRandom.current();  
    return lr.nextInt(1, 11);  
}

8. Given:

```
AtomicInteger i = new AtomicInteger();
```

Which atomically increment *i* by 9? (Choose all that apply.)

- A. i.addAndGet(9);
- B. i.getAndAdd(9);
- C. i.set(i.get() + 9);
- D. i.atomicIncrement(9);
- E. i = i + 9;

9. Given:

```
public class LeaderBoard {
 private ReadWriteLock rwl = new ReentrantReadWriteLock();
 private List<Integer> highScores = new ArrayList<>();
 public void addScore(Integer score) {
 // position A
 lock.lock();
 try {
 if (highScores.size() < 10) {
 highScores.add(score);
 } else if (highScores.get(highScores.size() - 1) < score) {
 highScores.set(highScores.size() - 1, score);
 } else {
 return;
 }
 Collections.sort(highScores, Collections.reverseOrder());
 } finally {
 lock.unlock();
 }
 }
 public List<Integer> getHighScores() {
 // position B
 lock.lock();
 try {
 return Collections.unmodifiableList(new ArrayList<>(highScores));
 } finally {
 lock.unlock();
 }
 }
}
```

Which block(s) of code best match the behavior of the methods in the LeaderBoard class? (Choose all that apply.)

- A. Lock lock = rwl.reentrantLock(); // should be inserted at position A
- B. Lock lock = rwl.reentrantLock(); // should be inserted at position B
- C. Lock lock = rwl.readLock(); // should be inserted at position A
- D. Lock lock = rwl.readLock(); // should be inserted at position B
- E. Lock lock = rwl.writeLock(); // should be inserted at position A
- F. Lock lock = rwl.writeLock(); // should be inserted at position B

**10.** Given:

```
ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
rwl.readLock().unlock();
System.out.println("READ-UNLOCK-1");
rwl.readLock().lock();
System.out.println("READ-LOCK-1");
rwl.readLock().lock();
System.out.println("READ-LOCK-2");
rwl.readLock().unlock();
System.out.println("READ-UNLOCK-2");
rwl.writeLock().lock();
System.out.println("WRITE-LOCK-1");
rwl.writeLock().unlock();
System.out.println("WRITE-UNLOCK-1");
```

What is the result?

- A. The code will not compile
- B. The code will compile and output:

READ-UNLOCK-1  
READ-LOCK-1  
READ-LOCK-2  
READ-UNLOCK-2

- C. The code will compile and output:

READ-UNLOCK-1  
READ-LOCK-1  
READ-LOCK-2  
READ-UNLOCK-2  
WRITE-LOCK-1  
WRITE-UNLOCK-1

- D. A `java.lang.IllegalMonitorStateException` will be thrown
11. Which class contains factory methods to produce preconfigured `ExecutorService` instances?
- A. `Executor`
  - B. `Executors`
  - C. `ExecutorService`
  - D. `ExecutorServiceFactory`

12. Given:

```
private Integer executeTask(ExecutorService service,
 Callable<Integer> task) {
 // insert here
}
```

Which set(s) of lines, when inserted, would correctly use the `ExecutorService` argument to execute the `Callable` and return the `Callable`'s result? (Choose all that apply.)

A. try {  
    return service.submit(task);  
} catch (Exception e) {  
    return null;  
}

B. try {  
    return service.execute(task);  
} catch (Exception e) {  
    return null;  
}

C. try {  
    Future<Integer> future = service.submit(task);  
    return future.get();  
} catch (Exception e) {  
    return null;  
}

D. try {  
    Result<Integer> result = service.submit(task);  
    return result.get();  
} catch (Exception e) {  
    return null;  
}

- 13.** Which are true? (Choose all that apply.)
- A. A Runnable may return a result but must not throw an Exception
  - B. A Runnable must not return a result nor throw an Exception
  - C. A Runnable must not return a result but may throw an Exception
  - D. A Runnable may return a result and throw an Exception
  - E. A Callable may return a result but must not throw an Exception
  - F. A Callable must not return a result nor throw an Exception
  - G. A Callable must not return a result but may throw an Exception
  - H. A Callable may return a result and throw an Exception

- 14.** Given:

```
public class IncrementAction extends RecursiveAction {
 private final int threshold;
 private final int[] myArray;
 private int start;
 private int end;
 public IncrementAction(int[] myArray, int start, int end, int threshold) {
 this.threshold = threshold;
 this.myArray = myArray;
 this.start = start;
 this.end = end;
 }
 @Override
 protected void compute() {
 if (end - start < threshold) {
 for (int i = start; i <= end; i++) {
 myArray[i]++;
 }
 } else {
 int midway = (end - start) / 2 + start;
 IncrementAction a1 = new IncrementAction(myArray, start,
 midway, threshold);
 IncrementAction a2 = new IncrementAction(myArray, midway + 1,
 end, threshold);
 // insert answer here
 }
 }
}
```

Which line(s), when inserted at the end of the `compute` method, would correctly take the place of separate calls to `fork()` and `join()`? (Choose all that apply.)

- A. `compute();`
- B. `forkAndJoin(a1, a2);`
- C. `computeAll(a1, a2);`
- D. `invokeAll(a1, a2);`

15. When writing a `RecursiveTask` subclass, which are true? (Choose all that apply.)

- A. `fork()` and `join()` should be called on the same task
- B. `fork()` and `compute()` should be called on the same task
- C. `compute()` and `join()` should be called on the same task
- D. `compute()` should be called before `fork()`
- E. `fork()` should be called before `compute()`
- F. `join()` should be called after `fork()` but before `compute()`

16. Given the following code fragment:

```
public static void sampleTest() {
 Stream<Integer> nums = Stream.of(10, 5, 3, 2);
 Optional<Integer> result =
 nums
 .parallel()
 .map(n -> n * 10)
 .reduce((n1, n2) -> n1 - n2);
 System.out.println("Result: " + result.get());
}
```

What is the result?

- A. 0

- B. 40
- C. The result is unpredictable
- D. Compilation fails
- E. An exception is thrown at runtime

**17.** Given the following code fragment:

```
Stream<List<String>> sDogNames =
 Stream.generate(() ->
 Arrays.asList("boi", "aiko", "charis", "zooey", "clover"))
 .limit(2).unordered();

 sDogNames.parallel()
 .flatMap(s -> s.stream())
 .map(s -> s.toUpperCase())
 .forEach(s -> System.out.print(s + " "));
```

What is the result?

- A. BOI AIKO CHARIS ZOOEY CLOVER BOI AIKO CHARIS ZOOEY CLOVER
- B. Most likely, although not guaranteed: BOI AIKO CHARIS ZOOEY CLOVER  
BOI AIKO CHARIS ZOOEY CLOVER
- C. A ConcurrentModificationException is thrown
- D. Compilation fails
- E. An exception other than ConcurrentModificationException is thrown at runtime

**18.** Given the following code fragment:

```

Stream<List<String>> sDogNames2 =
 Arrays.asList(
 Arrays.asList("boi", "aiko", "charis", "zooey", "clover"),
 Arrays.asList("boi", "aiko", "charis", "zooey", "clover"))
 .stream().unordered();
 sDogNames2.parallel()
 .flatMap(s -> s.stream())
 .map(s -> s.toUpperCase())
 .forEach(s -> System.out.print(s + " "));

```

What is the result?

- A. BOI AIKO CHARIS ZOOEY CLOVER BOI AIKO CHARIS ZOOEY CLOVER
  - B. The result is unpredictable
  - C. A ConcurrentModificationException is thrown
  - D. Compilation fails
  - E. An exception is thrown at runtime
- 19.** Given the code fragment:

```

List<Integer> myNums = Arrays.asList(1, 2, 3, 4, 5);
OptionalInt aNum = myNums.parallelStream().mapToInt(i -> i * 2).findAny();
aNum.ifPresent(System.out::println);

```

What is the result?

- A. 1
- B. 2
- C. The result is unpredictable; it could be any one of the numbers in the stream
- D. A ConcurrentModificationException is thrown
- E. Compilation fails
- F. An exception is thrown at runtime

# A SELF TEST ANSWERS

1.  **C** is correct. The `Iterator` is obtained before 6 is added. As long as the reference to the `Iterator` is maintained, it will only provide access to the values 4 and 2.  
 **A, B, D, E, and F** are incorrect based on the above. (OCP Objective 10.4)
2.  **C** is correct. Because the `Iterator` is obtained before `remove()` is invoked, it will reflect all the elements that have been added to the collection.  
 **A, B, D, E, and F** are incorrect based on the above. (OCP Objective 10.4)
3.  **A** and **D** are correct. Of the methods listed, only `add` and `remove` will modify the list and cause a new internal array to be created.  
 **B** and **C** are incorrect based on the above. (OCP Objective 10.4)
4.  **C** is correct. The `add` method will throw an `IllegalStateException` if the queue is full. The two `offer` methods will return `false` if the queue is full. Only the `put` method will block until space becomes available.  
 **A, B, and D** are incorrect based on the above. (OCP Objective 10.4)
5.  **C** is correct; you could see the output in either order because we don't know in advance if threads `t1`, `t2`, and `t3` will complete before the main thread or if the main thread will complete first. However, in either case, `t1`, `t2`, and `t3` will wait at the barrier until `t1`, `t2`, and `t3` are all at the barrier, and the last thread to reach the barrier will display the output.  
 **A, B, D, and E** are incorrect; **A** and **B** are incorrect because of the above. **D** is incorrect because the `Singleton` is thread-safe; `enum` singletons are guaranteed to be created in a thread-safe manner, and we have synchronized the `updateValue()` method. **E** is incorrect because it is highly unlikely one of these threads will get stuck or time out and none is accessing any collection that is not thread-safe. (OCP Objective 10.4)
6.  **C** is correct; it uses the correct syntax.  
 The methods for answers **A, B, and D** do not exist in a `ConcurrentHashMap`. A traditional `Map` contains a single-argument `remove` method that removes an element based on its key. The `ConcurrentMap` interface (which `ConcurrentHashMap` implements) added the two-argument

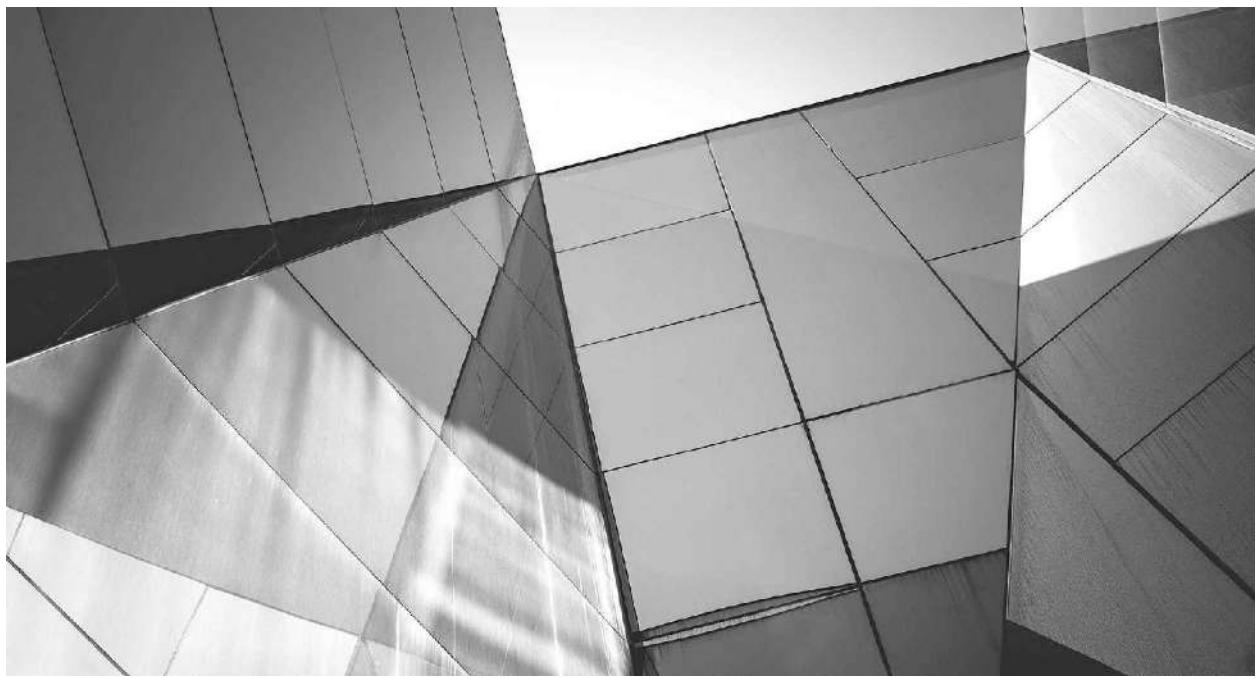
remove method, which takes a key and a value. An element will only be removed from the Map if its value matches the second argument. A boolean is returned to indicate if the element was removed. (OCP Objective 10.4)

7.  **D** is correct. The `ThreadLocalRandom` creates and retrieves `Random` instances that are specific to a thread. You could achieve the same effect prior to Java 7 by using the `java.lang.ThreadLocal` and `java.util.Random` classes, but it would require several lines of code. `Math.random()` is thread-safe, but uses a shared `java.util.Random` instance and can suffer from contention problems.  
 **A, B, and C** are incorrect based on the above. (OCP Objective 10.1)
8.  **A** and **B** are correct. The `addAndGet` and `getAndAdd` both increment the value stored in an `AtomicInteger`.  
 Answer **C** is not atomic because in between the call to `get` and `set`, the value stored by `i` may have changed. Answer **D** is invalid because the `atomicIncrement` method is fictional, and answer **E** is invalid because auto-boxing is not supported for the atomic classes. The difference between the `addAndGet` and `getAndAdd` methods is that the first is a prefix method (`++x`) and the second is a postfix method (`x++`). (Objective 10.3)
9.  **D** and **E** are correct. The `addScore` method modifies the collection and, therefore, should use a write lock, whereas the `getHighScores` method only reads the collection and should use a read lock.  
 **A, B, C, and F** are incorrect; they will not behave correctly. (Objective 10.3)
10.  **D** is correct. A lock counts the number of times it has been locked. Calling `lock` increments the count, and calling `unlock` decrements the count. If a call to `unlock` decreases the count below zero, an exception is thrown.  
 **A, B, and C** are incorrect based on the above. (OCP Objective 10.3)
11.  **B** is correct. `Executor` is the super-interface for `ExecutorService`. You use `Executors` to easily obtain `ExecutorService` instances with predefined threading behavior. If the `Executor` interface does not produce `ExecutorService` instances with the behaviors that you desire, you can always look into using `java.util.concurrent.AbstractExecutorService` or `java.util.concurrent.ThreadPoolExecutor` directly.  
 **A, C, and D** are incorrect based on the above. (OCP Objective 10.1)

- 12.**  **C** is correct. When you submit a `Callable` to an `ExecutorService` for execution, you will receive a `Future` as the result. You can use the `Future` to check on the status of the `Callable`'s execution, or just use the `get()` method to block until the result is available.
- A, B, and D** are incorrect based on the above. (OCP Objective 10.1)
- 13.**  **B** and **H** are correct. `Runnable` and `Callable` serve similar purposes. `Runnable` has been available in Java since version 1. `Callable` was introduced in Java 5 and serves as a more flexible alternative to `Runnable`. A `Callable` allows a generic return type and permits thrown exceptions, whereas a `Runnable` does not.
- A, C, D, E, F, and G** are incorrect statements. (Objective 10.1)
- 14.**  **D** is correct. The `invokeAll` method is a var-args method that will fork all Fork/Join tasks, except one that will be invoked directly.
- A, B, and C** are incorrect; they would not correctly complete the Fork/Join process. (OCP Objective 10.5)
- 15.**  **A and E** are correct. When creating multiple `ForkJoinTask` instances, all tasks except one should be forked first, so they can be picked up by other Fork/Join worker threads. The final task should then be executed within the same thread (typically by calling `compute()`) before calling `join()` on all the forked tasks to await their results. In many cases, calling the methods in the wrong order will not result in any compiler errors, so care must be taken to call the methods in the correct order.
- B, C, D, and F** are incorrect based on the above. (OCP Objective 10.5)
- 16.**  **C** is correct. The result is unpredictable because the reduction function is not associative and the stream is parallel.
- A, B, D, and E** are incorrect based on the above (OCP Objective 10.6)
- 17.**  **B** is correct. Because of the `limit(2)`, we will see the dog names twice, most likely in order. The `limit()` forces the stream to maintain state about the source data, so it is likely the stream will retain the original source ordering (although that is not guaranteed) even though it's an unordered parallel stream.
- A, C, D, and E** are incorrect based on the above (OCP Objective 10.6)
- 18.**  **B** is correct. Unlike the previous sample question, we have no `limit()` so

we have a stateless stream pipeline. The stream is unordered, and the pipeline is executing in parallel, so we cannot predict the ordering of the output. Note that if we were using `forEachOrdered()` instead of `forEach()` as the terminal operation, it is likely, but not guaranteed, that we would see the results in the same ordering as the source.

- A, C, D, and E** are incorrect based on the above (OCP Objective 10.6)
- 19.**  **C** is correct. For a parallel stream, any of the numbers can be multiplied by 2 and returned by `findAny()`.
- A, B, D, E, and F** are incorrect based on the above (OCP Objective 10.6)



# 12

## JDBC

### CERTIFICATION OBJECTIVES

- Describe the interfaces that Make Up the Core of the JDBC API Including the Driver, Connection, Statement, and ResultSet Interfaces and Their Relationship to Provider Implementations
- Identify the Components Required to Connect to a Database Using the DriverManager Class Including the JDBC URL
- Submit Queries and Read Results from the Database Including Creating Statements, Returning Result Sets, Iterating Through the Results, and Properly Closing Result Sets, Statements, and Connections



Two-Minute Drill

**Q&A** Self Test

This chapter covers the JDBC API that was added for the Java SE 7 and 8 exams. The exam developers have long felt that this API is truly a core feature of the language, and being able to demonstrate proficiency with JDBC goes a long way toward demonstrating your skills as a Java programmer.

Interestingly, JDBC has been a part of the language since JDK version 1.1 (1997) when JDBC 1.0 was introduced. Since then, there has been a steady progression of updates to the API, roughly one major release for each even-numbered JDK release, with the last major update being JDBC 4.0, released in 2006 with Java SE 6. In Java SE 7 and 8, JDBC got some minor updates and is now at version 4.2. While the focus of the exam is on JDBC 4.x, there may be questions about the differences between loading a driver with a JDBC 3.0 and

JDBC 4.x implementation, so we'll talk about that as well.

The good news is that the exam is not going to test your ability to write SQL statements. That would be an exam all by itself (maybe even more than one—SQL is a BIG topic!). But you will need to recognize some basic SQL syntax and commands, so we'll start by spending some time covering the basics of relational database systems and give you enough SQL to make you popular at database parties. If you feel you have experience with SQL and understand database concepts, you might just skim the first section or skip right to the first exam objective and dive right in.

## Starting Out: Introduction to Databases and JDBC

When you think of organizing information and storing it in some easily understood way, a spreadsheet or a table is often the first approach you might take. A spreadsheet or a table is a natural way of categorizing information: The first row of a table defines the sort of information that the table will hold, and each subsequent row contains a set of data that is related to the key we create on the left. For example, suppose you wanted to chart your monthly spending for several types of expenses ([Table 12-1](#)).

**TABLE 12-1** Methods to Map, Filter, and Reduce

| Month    | Gas      | EatingOut | Utilities | Phone   |
|----------|----------|-----------|-----------|---------|
| January  | \$200.25 | \$109.87  | \$97.00   | \$45.08 |
| February | \$225.34 | \$121.08  | \$97.00   | \$23.36 |
| March    | \$254.78 | \$130.45  | \$97.00   | \$56.09 |

From the data in the chart, we can determine that your overall expenses are increasing month to month in the first three months of this year. But notice that without the table, without a relationship between the month and the data in the columns, you would just have a pile of receipts with no way to draw out

important conclusions, such as

- Assuming you drove the same number of miles per month, gas is getting pricey—maybe it is time to get a Prius.
- You are eating out more month to month (or the price of eating out is going up)—maybe it's time to start doing some meal planning.
- And maybe you need to be a little less social—that phone bill is high.

The point is that this small sample of data is the key to understanding a relational database system. A relational database is really just a software application designed to store and manipulate data in tables. The software itself is actually called a Relational Database Management System (RDBMS), but many people shorten that to just “database”—so know that going forward, when we refer to a database, we are actually talking about an RDBMS (the whole system). What the relational management system adds to a database is the ability to define relationships between tables. It also provides a language to get data in and out in a meaningful way.

Looking at the simple table in [Table 12-1](#), we know that the data in the columns, Gas, EatingOut, Utilities, and Phone, are grouped by the months January, February, and so on. The month is unique to each row and identifies this row of data. In database parlance, the month is a “primary key.” A primary key is generally required for a database table to identify which row of the table you want and to make sure that there are no duplicate rows.

Extending this a little further, if the data in [Table 12-1](#) were stored in a database, I could ask the database (write a query) to give me all of the data for the month of January (again, my primary key is “month” for this table). I might write something like:

“Give me all of my expenses for January.”

The result would be something like:

January: Gas: \$200.25, EatingOut: \$109.87, Utilities: \$97.00, Phone: \$45.08

This kind of query is what makes a database so powerful. With a relatively simple language, you can construct some really powerful queries in order to manipulate your data to tell a story. In most RDBMSs, this language is called the Structured Query Language (SQL). The same query we wrote out in a sentence earlier would be expressed like this in SQL:

```
SELECT * FROM Expenses WHERE Month = 'January'
```

which can be translated to “select all of the columns (\*) from my table named ‘Expenses’ where the month column is equal to the string ‘January’.” Let’s look a bit more at how we “talk” to a database and what other sorts of queries we can make with tables in a relational database.

## Talking to a Database

There are three important concepts when working with a database:

- Creating a connection to the database
- Creating a statement to execute in the database
- Getting back a set of data that represents the results

Let’s look at these concepts in more detail.

Before we can communicate with the software that manages the database, before we can send it a query, we need to make a connection with the RDBMS itself. There are many different types of connections, and a lot of underlying technology to describe the connection itself, but in general, to communicate with an RDBMS, we need to open a connection using an IP address and port number to the database. Once we have established the connection, we need to send it some parameters (such as a username and password) to authenticate ourselves as a valid user of the RDBMS. Finally, assuming all went well, we can send queries through the connection. This is like logging into your online account at a bank. You provide some credentials, a username and password, and a connection is established and opened between you and the bank. Later in the chapter, when we start writing code, we’ll open a connection using a Java class called the `DriverManager`, and in one request, pass in the database name, our username, and password.

Once we have established a connection, we can use some type of application (usually provided by the database vendor) to send query statements to the database, have them executed in the database, and get a set of results returned. A set of results can be one row, as we saw before when we asked for the data from the month of January, or several rows. For example, suppose we wanted to see all of the Gas expenses from our Expenses table. We might query the database like this:

"Show me all of my Gas Expenses"

Or as a SQL query:

```
SELECT Gas FROM Expenses
```

The set of results that would "return" from my query would be three rows, and each row would contain one column.

|          |
|----------|
| \$200.25 |
| \$225.34 |
| \$254.78 |

An important aspect of a database is that the data is presented back to you exactly the same way that it is stored. Since Gas expense is a column, the query will return three rows (one for January, one for February, and one for March). Note that because we did not ask the database to include the Month column in the results, all we got was the Gas column. The results do preserve the fact that Gas is a column and not a row and, in general, present the data in the same row-and-column order in which it is stored in the database.

## SQL Queries

Let's look a bit more at the syntax of SQL, the language used to write queries in a database. There are really four basic SQL queries that we are going to use in this chapter and that are common to manipulating data in a database. In summary, the SQL commands we are interested in are used to perform CRUD operations.

Like most terms presented in all caps, CRUD is an acronym and means *Create, Read, Update, and Delete*. These are the four basic operations for data in a database. They are represented by four distinct SQL commands, detailed in [Table 12-2](#).

**TABLE 12-2** Example SQL CRUD Commands

| "CRUD"         | SQL Command | Example SQL Query                                                                 | Expressed in English                                                    |
|----------------|-------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------|
| Create         | INSERT      | INSERT INTO Expenses<br>VALUES ('April', 231.21,<br>29.87, 97.00, 45.08)          | Add a new row (April) to<br>expenses with the following<br>values....   |
| Read (or Find) | SELECT      | SELECT * FROM Expenses<br>WHERE Month="February"                                  | Get me all of the columns<br>in the Expenses table for<br>February.     |
| Read All       | SELECT      | SELECT * FROM Expenses                                                            | Get me all of the columns in<br>the Expenses table.                     |
| Update         | UPDATE      | UPDATE Expenses<br>SET Phone=32.36,<br>EatingOut=111.08<br>WHERE Month='February' | Change my Phone expense<br>and EatingOut expense for<br>February to.... |
| Delete         | DELETE      | DELETE FROM Expenses<br>WHERE Month='April'                                       | Remove the row of expenses<br>for April.                                |

Here is a quick explanation for the examples in [Table 12-2](#):

- **INSERT** Add a row to the table Expenses, and set each of the columns in the table to the values expressed in the parentheses.
- **SELECT with WHERE** You have already seen the SELECT clause with a WHERE clause, so you know that this SQL statement returns a single row identified by the primary key—the Month column. Think of this statement as a refinement to Read—more like a Find or Find by primary key.

- **SELECT** When the SELECT clause does not have a WHERE clause, we are asking the database to return every row. Further, because we are using an asterisk (\*) following the SELECT, we are asking for every column. Basically, it is a dump of the data shown in Table 15-1. Think of this statement as a Read All.
- **UPDATE** Change the data in the Phone and EatingOut cells to the new data provided for February.
- **DELETE** Remove a row altogether from the database where the Month is April.

Really, this is all the SQL you need to know for this chapter. There are many other SQL commands, but this is the core set. If we need to go beyond this set of four commands in the chapter, we will cover them as they come up. Now, let's look at a more detailed database example that we will use as the example set of tables for this chapter, using the data requirements of a small bookseller, Bob's Books.



*SQL commands, like SELECT, INSERT, UPDATE, and so on, are case-insensitive. So it is largely by convention (and one we will use in this chapter) that we use all capital letters for SQL commands and key words, such as WHERE, FROM, LIKE, INTO, SET, and VALUES. SQL table names and column names, also called identifiers, can be case-sensitive or case-insensitive, depending on the database. The example code shown in this chapter uses a case-insensitive database, so again, just for convention, we will use upper camel case, that is, the first letter of each noun capitalized and the rest in lowercase.*

*One final note about case—all databases preserve case when a string is delimited—that is, when it is enclosed in quotes. So a SQL clause that uses single or double quotation marks to delimit an identifier will preserve the case of the identifier.*

## Bob's Books, Our Test Database

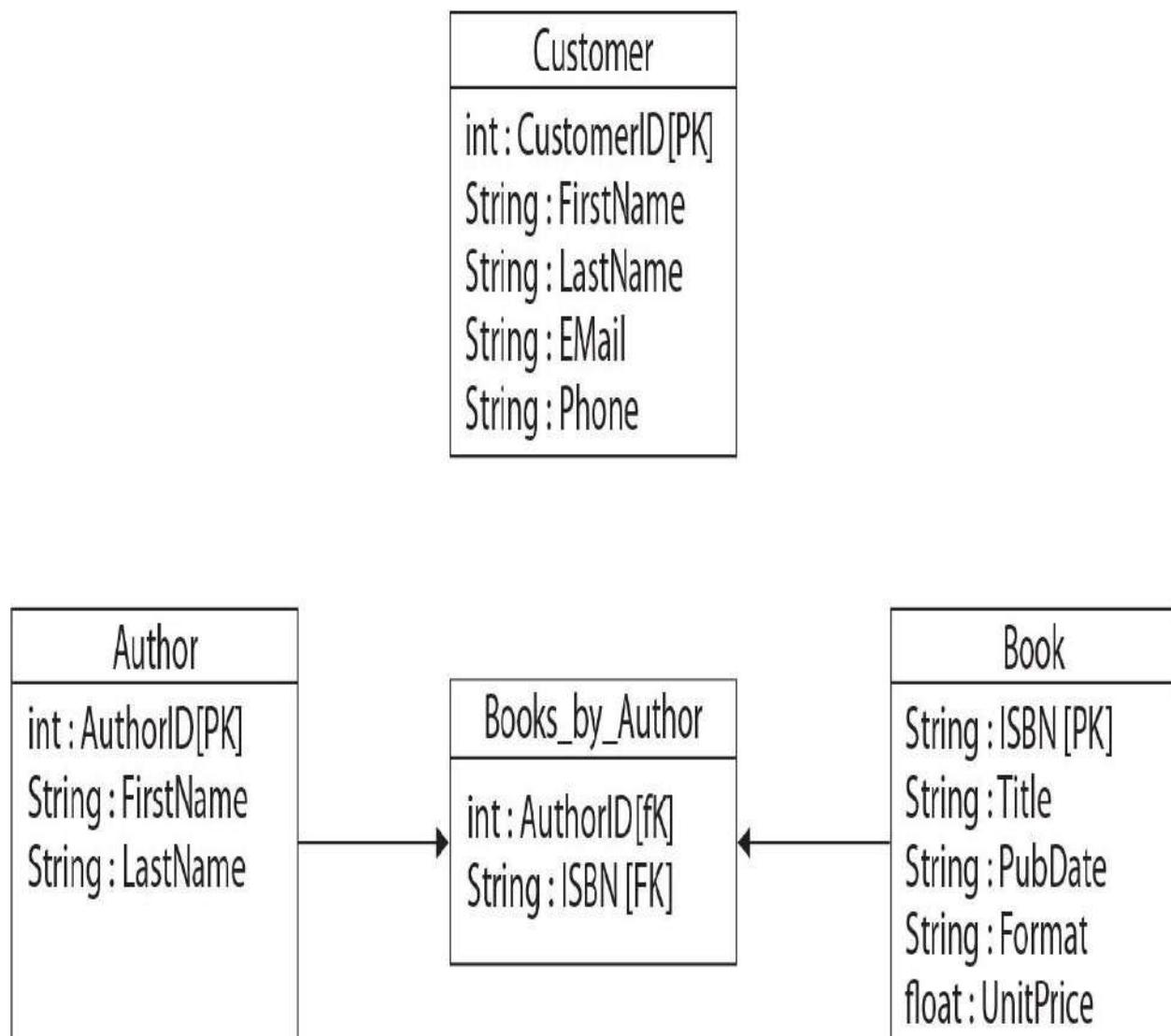
In this section, we'll describe a small database with a few tables and a few rows

of data. As we work through the various JDBC topics in this chapter, we'll work with this database.

Bob is a small bookseller who specializes in children's books. Bob has designed his data around the need to sell his books online using a database (which one doesn't really matter) and a Java application. Bob has decided to use the JDBC API to allow him to connect to a database and perform queries through a Java application.

To start, let's look at the organization of Bob's data. In a database, the organization and specification of the tables is called the database schema ([Figure 12-1](#)). Bob's is a relatively simple schema, and again, for the purposes of this chapter, we are going to concentrate on just four tables from Bob's schema.

**FIGURE 12-1** Bob's BookSeller database schema



This is a relatively simple schema that represents a part of the database for a small bookstore. In the schema shown, there is a table for Customer (Table 12-3). This table stores data about Bob's customers—a customer ID, first name and last name, an e-mail address, and phone number. Postal addresses and other information could be stored in another table.

**TABLE 12-3** Bob's Books Customer Table Sample Data

e

| CustomerID | FirstName   | LastName  | Email                    | Phone        |
|------------|-------------|-----------|--------------------------|--------------|
| 5000       | John        | Smith     | john.smith@verizon.net   | 555-340-1230 |
| 5001       | Mary        | Johnson   | mary.johnson@comcast.net | 555-123-4567 |
| 5002       | Bob         | Collins   | bob.collins@yahoo.com    | 555-012-3456 |
| 5003       | Rebecca     | Mayer     | rebecca.mayer@gmail.com  | 555-205-8212 |
| 5006       | Anthony     | Clark     | anthony.clark@gmail.com  | 555-256-1901 |
| 5007       | Judy        | Sousa     | judy.sousa@verizon.net   | 555-751-1207 |
| 5008       | Christopher | Patriquin | patriquinc@yahoo.com     | 555-316-1803 |
| 5009       | Deborah     | Smith     | debsmith@comcast.net     | 555-256-3421 |
| 5010       | Jennifer    | McGinn    | jmcginn@comcast.net      | 555-250-0918 |

The next three tables we will look at represent the data required to store information about books that Bob sells. Because a book is a more complex set of data than a customer, we need to use one table for information about books, one for information about authors, and a third to create a relationship between books and authors.

Suppose that you tried to store a book in a single table with a column for the

ISBN (International Standard Book Number), title, and author name. For many books, this would be fine. But what happens if a book has two authors? Or three authors? Remember that one requirement for a database table is a unique primary key, so you can't simply repeat the ISBN in the table. In fact, having two rows with the same primary key will violate a key constraint in relational database design: the primary key of every row must be unique.

| ISBN | Title                | Author         |
|------|----------------------|----------------|
| ABCD | The Wonderful Life   | Fred Smith     |
| ABCD | The Wonderful Life   | Tom Jones      |
| 1234 | Some Enchanted Night | Paula Fredrick |

Instead, there needs to be a way to have a separate table of books and authors and some way to link them together. Bob addressed this issue by placing Books in one table ([Table 12-4](#)) and Authors ([Table 12-5](#)) in another. The primary key for Books is the ISBN number, and therefore, each Book entry will be unique. For the Author table, Bob is creating a unique AuthorID for each author in the table.

**TABLE 12-4** Bob's Books Sample Data for the "Books" Table

| ISBN       | Title                                        | PubDate    | Format    | Price |
|------------|----------------------------------------------|------------|-----------|-------|
| 142311339X | The Lost Hero<br>(Heroes of Olympus, Book 1) | 2010-10-12 | Hardcover | 10.95 |
| 0689852223 | The House of the Scorpion                    | 2002-01-01 | Hardcover | 16.95 |
| 0525423656 | Crossed (Matched Trilogy, Book 2)            | 2011-11-01 | Hardcover | 12.95 |
| 1423153627 | The Kane Chronicles Survival Guide           | 2012-03-01 | Hardcover | 13.95 |
| 0439371112 | Howliday Inn                                 | 2001-11-01 | Paperback | 14.95 |
| 0439861306 | The Lightning Thief                          | 2006-03-12 | Paperback | 11.95 |
| 031673737X | How to Train Your Dragon                     | 2010-02-01 | Hardcover | 10.95 |
| 0545078059 | The White Giraffe                            | 2008-05-01 | Paperback | 6.95  |
| 0803733428 | The Last Leopard                             | 2009-03-05 | Hardcover | 13.95 |
| 9780545236 | Freaky Monday                                | 2010-01-15 | Paperback | 12.95 |

**TABLE 12-5** Bob's Books Author Table Sample Data for the “Authors” Table

| AuthorID | FirstName | LastName |
|----------|-----------|----------|
| 1000     | Rick      | Riordan  |
| 1001     | Nancy     | Farmer   |
| 1002     | Ally      | Condie   |
| 1003     | Cressida  | Cowell   |
| 1004     | Lauren    | St. John |
| 1005     | Eoin      | Colfer   |
| 1006     | Esther    | Freisner |
| 1007     | Chris     | D'lacey  |
| 1008     | Mary      | Rodgers  |
| 1009     | Heather   | Hatch    |

To tie Authors to Books and Books to Authors, Bob has created a third table called Books\_by\_Author. This is a unique table type in a relational database. This table is called a *join* table. In a join table, there are no primary keys—instead, all the columns represent data that can be used by other tables to create a relationship. These columns are referred to as foreign keys—they represent a primary key in another table. Looking at the last two rows of this table, you can see that the Book with the ISBN 9780545236 has two authors: author id 1008 (Mary Rodgers) and 1009 (Heather Hatch). Using this join table, we can combine the two sets of data without needing duplicate entries in either table. We'll return to the concept of a join table later in the chapter.

**TABLE 12-6** Bob's Books Books by Author Sample Data

| AuthorID | ISBN       |
|----------|------------|
| 1000     | 142311339X |
| 1001     | 0689852223 |
| 1002     | 0525423656 |
| 1000     | 1423153627 |
| 1003     | 031673737X |
| 1004     | 0545078059 |
| 1004     | 0803733428 |
| 1008     | 9780545236 |
| 1009     | 9780545236 |

A complete Bob's Books database schema would include tables for publishers, addresses, stock, purchase orders, and other data that the store needs to run its business. But for our purposes, this part of the schema is sufficient. Using this schema, we can write SQL queries using the SQL CRUD commands you learned earlier.

To summarize, before looking at JDBC, you should now know about connections, statements, and result sets:

- A connection is how an application communicates with a database.
- A statement is a SQL query that is executed on the database.
- A result set is the data that is returned from a SELECT statement.

Having these concepts down, we can use Bob's Books simple schema to frame some common uses of the JDBC API to submit SQL queries and get results in a

Java application.

## CERTIFICATION OBJECTIVE

### Core Interfaces of the JDBC API (OCP Objective 11.1)

*11.1 Describe the interfaces that make up the core of the JDBC API including the Driver, Connection, Statement, and ResultSet interfaces and their relationship to provider implementations.*

As we mentioned in the previous section, the purpose of a relational database is really threefold:

- To provide storage for data in tables
- To provide a way to create relationships between the data—just as Bob did with the Authors, Books, and Books\_by\_Author tables
- To provide a language that can be used to get the data out, update the data, remove the data, and create new data

The purpose of JDBC is to provide an application programming interface (API) for Java developers to write Java applications that can access and manipulate relational databases and use SQL to perform CRUD operations.

Once you understand the basics of the JDBC API, you will be able to access a huge list of databases. One of the driving forces behind JDBC was to provide a standard way to access relational databases, but JDBC can also be used to access file systems and object-oriented data sources. The key is that the API provides an abstract view of a database connection, statements, and result sets. These concepts are represented in the API as interfaces in the `java.sql` package: `Connection`, `Statement`, and `ResultSet`, respectively. What these interfaces define are the *contracts* between you and the implementing class. In truth, you may not know (nor should you care) *how* the implementation class works. As long as the implementation class implements the interface you need, you are assured that the methods defined by the interface exist and you can invoke them.

The `java.sql.Connection` interface defines the contract for an object that represents the connection with a relational database system. Later, we will look at the methods of this contract, but for now, an instance of a `Connection` is what

we need to communicate with the database. How the `Connection` interface is implemented is vendor dependent, and again, we don't need to worry so much about the how—as long as the vendor follows the contract, we are assured that the object that represents a `Connection` will allow us to work with a database connection.

The `Statement` interface provides an abstraction of the functionality needed to get a SQL statement to execute on a database, and a `ResultSet` interface is an abstraction functionality needed to process a result set (the table of data) that is returned from the SQL query when the query involves a SQL SELECT statement.

The classes that implement `Connection`, `Statement`, `ResultSet`, and a number of other interfaces we will look at shortly are created by the vendor of the database we are using. The vendor understands their database product better than anyone else, so it makes sense that they create these classes. And it allows the vendor to optimize or hide any special characteristics of their product. The collection of the implementation classes is called the JDBC driver. A JDBC driver (lowercase “d”) is the collection of classes required to support the API, whereas `Driver` (uppercase “D”) is one of the implementations required in a driver.

A JDBC driver is typically provided by the vendor in a JAR or ZIP file. The implementation classes of the driver must meet a minimum set of requirements in order to be JDBC compliant. The JDBC specification provides a list of the functionality that a vendor must support and what functionality a vendor may optionally support.

Here is a partial list of the requirements for a JDBC driver. For more details, please read the specification (JSR-221). Note that the details of implementing a JDBC driver are NOT on the exam.

- Fully implement the interfaces: `java.sql.Driver`,  
`java.sql.DatabaseMetaData`, `java.sql.ResultSetMetaData`.
- Implement the `java.sql.Connection` interface. (Note that some methods are optional depending on the SQL version the database supports—more on SQL versions later in the chapter.)
- Implement `java.sql.Statement`, and `java.sql.PreparedStatement`.
- Implement the `java.sql.CallableStatement` interfaces if the database supports stored procedures. Again, more on this interface later in the chapter.

- Implement the `java.sql.ResultSet` interface.

## CERTIFICATION OBJECTIVE

### Connect to a Database Using DriverManager (OCP Objective 11.2)

*11.2 Identify the components required to connect to a database using the DriverManager class including the JDBC URL*

Not all of the types defined in the JDBC API are interfaces. One important class for JDBC is the `java.sql.DriverManager` class. This concrete class is used to interact with a JDBC driver and return instances of connection objects to you. Conceptually, the way this works is by using a Factory design pattern. Next, we'll look at `DriverManager` in more detail.



*Let's take this opportunity to see the factory design pattern in use. In a factory pattern, a concrete class with static methods is used to create instances of objects that implement an interface. For example, suppose we wanted to create an instance of a `Vehicle` object:*

```
public interface Vehicle {
 public void start(); // Methods we think all vehicles should
 public void stop(); // support.
}
```

*We need an implementation of `Vehicle` in order to use this contract. So we design a `Car`:*

```
package com.us.automobile;
public class Car implements Vehicle {
 public void start() { } // ... do start things
 public void stop() { } // ... do stop things
}
```

***In order to use the car, we could create one:***

```
public class MyClass {
 public static void main(String args[]) {
 Vehicle ferrari =
 new com.us.automobile.Car(); // Create a Ferrari
 ferrari.start(); // Start the Ferrari
 }
}
```

***However, here it would be better to use a factory—that way, we need not know anything about the actual implementation, and, as we will see later with DriverManager, we can use methods of the factory to dynamically determine which implementation to use at runtime.***

```
public class MyClass {
 public static void main(String args[]) {
 Vehicle ferrari =
 CarFactory.getVehicle("Ferrari"); // Use a factory to
 // create a Ferrari
 ferrari.start();
 }
}
```

***The factory in this case could create a different car based on the string passed to the static getVehicle() method—something like this:***

```
public class CarFactory {
 public static Vehicle getVehicle(String type) {
 // ... create an instance of an object that represents the
 // type of car passed as the argument
 }
}
```

***DriverManager uses this factory pattern to “construct” an instance of a Connection object by passing a string to its getConnection() method.***

## The DriverManager Class

The `DriverManager` class is a concrete, utility class in the JDBC API with static methods. You will recall that static or class methods can be invoked by other classes using the class name. One of those methods is `getConnection()`, which we look at next.

The `DriverManager` class is so named because it manages which JDBC driver implementation you get when you request an instance of a `Connection` through the `getConnection()` method.

There are several overloaded `getConnection` methods, but they all share one common parameter: a `String` URL. One pattern for `getConnection` is

```
DriverManager.getConnection(String url, String username, String password);
```

For example:

```

String url
 = "jdbc:derby://localhost:1521/BookSellerDB"; // JDBC URL
String user = "bookguy"; // BookSellerDB user name
String pwd = "$3lleR"; // BookSellerDB password
try {
 Connection conn
 = DriverManager.getConnection(url, user, pwd); // Get an
 // instance of a
 // Connection
 // object
} catch (SQLException se) { }

```

In this example, we are creating a connection to a Derby database, on a network, at a localhost address (on the local machine), at port number 1521, to a database called "BookSellerDB", and we are using the credentials "bookguy" as the user id, and "\$3lleR" as the password. Don't worry too much about the syntax of the URL right now—we'll cover that soon.



***It's a horrible idea to hard-code a username and password in the getconnection() method. Obviously, anyone reading the code would then know the username and password to the database. A more secure way to handle database credentials would be to separate the code that produces the credentials from the code that makes the connection. In some other class, you would use some type of authentication and authorization code to produce a set of credentials to allow access to the database. For simplicity in the examples in the chapter, we'll hard-code the username and password, but just keep in mind that on the job, this is not a best practice.***

When you invoke the `DriverManager`'s `getConnection()` method, you are

asking the `DriverManager` to try passing the first string in the statement, the driver URL, along with the username and password to each of the driver classes registered with the `DriverManager` in turn. If one of the driver classes recognizes the URL string, and the username and password are accepted, the driver returns an instance of a `Connection` object. If, however, the URL is incorrect, or the username and/or password are incorrect, then the method will throw a `SQLException`. We'll spend some time looking at `SQLException` later in this chapter.

## How JDBC Drivers Register with the `DriverManager`

Because this part of the JDBC process is important to understand, and it involves a little Java magic, let's spend some time diagramming how driver classes become "registered" with the `DriverManager`, as shown in [Figure 12-2](#).

**FIGURE 12-2**

How JDBC drivers self-register with `DriverManager`

Start your application:

```
java -classpath ... MyDBApp
```



DriverManager  
(factory)

Classload the class defined in the  
META-INF/services/java.sql.Driver file.

```
DriverManager.registerDriver(this);
```



A JDBC driver  
(jar file)

Repeat this process for every  
jar file in the classpath that has  
a java.sql.Driver file.

First, one or more JDBC drivers, in a JAR or ZIP file, are included in the classpath of your application. The `DriverManager` class uses a service provider mechanism to search the classpath for any JAR or ZIP files that contain a file named `java.sql.Driver` in the `META-INF/services` folder of the driver jar or zip. This is simply a text file that contains the full name of the class that the vendor used to implement the `jdbc.sql.Driver` interface. For example, for a Derby driver, the full name is `org.apache.derby.jdbc.ClientDriver`.

The `DriverManager` will then attempt to load the class it found in the `java.sql.Driver` file using the class loader:

```
Class.forName("org.apache.derby.jdbc.ClientDriver");
```

When the driver class is loaded, its static initialization block is executed. Per the JDBC specification, one of the first activities of a driver instance is to “self-register” with the `DriverManager` class by invoking a static method on

`DriverManager`. The code (minus error handling) looks something like this:

```
public class ClientDriver implements java.sql.Driver{
 static {
 ClientDriver driver = new ClientDriver();
 DriverManager.registerDriver(driver);
 }
 //...
}
```

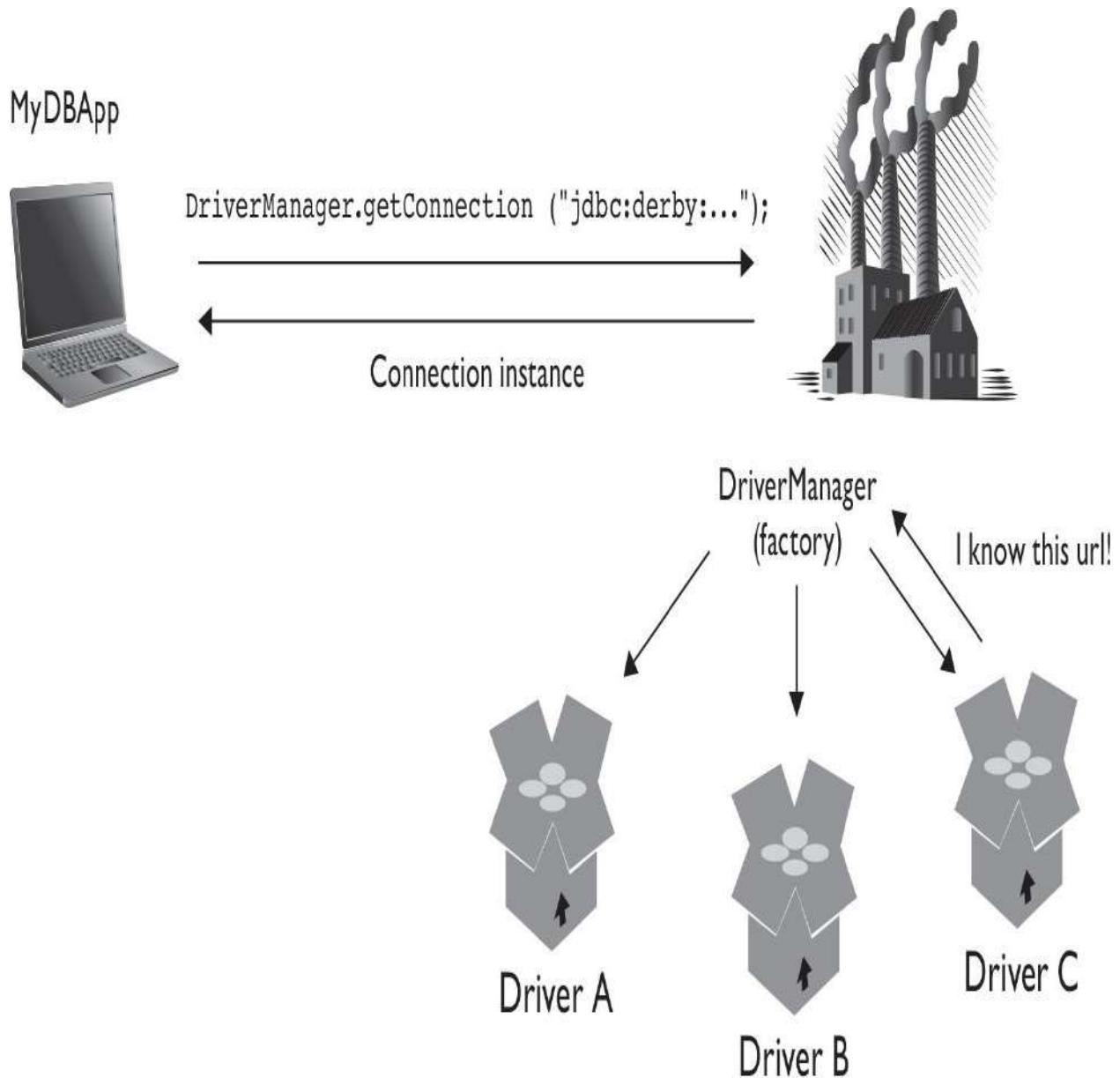
This registers (stores) an instance of the `Driver` class into the `DriverManager`.

Now, when your application invokes the `DriverManager.getConnection()` method and passes a JDBC URL, username, and password to the method, the `DriverManager` simply invokes the `connect()` method on the registered `Driver`. If the connection was successful, the method returns a `Connection` object instance to `DriverManager`, which, in turn, passes that back to you.

If there is more than one registered driver, the `DriverManager` calls each of the drivers in turn and attempts to get a `Connection` object from them, as shown in [Figure 12-3](#).

### **FIGURE 12-3**

How the `DriverManager` gets a Connection



Pass the url, name, and password to each of the registered drivers in turn until one returns a non-null Connection.

The first driver that recognizes the JDBC URL and successfully creates a connection using the username and password will return an instance of a `Connection` object. If no drivers recognize the URL, username, and password combination, or if there are no registered drivers, then a `SQLException` is thrown.

instead.

To summarize:

- The JVM loads the `DriverManager` class, a concrete class in the JDBC API.
- The `DriverManager` class loads any instances of classes it finds in the `META-INF/services/java.sql.Driver` file of JAR/ZIP files on the classpath.
- Driver classes call `DriverManager.register(this)` to self-register with the `DriverManager`.
- When the `DriverManager.getConnection(String url)` method is invoked, `DriverManager` invokes the `connect()` method of each of these registered `Driver` instances with the URL string.
- The first `Driver` that successfully creates a connection with the URL returns an instance of a `Connection` object to the `DriverManager.getConnection` method invocation.

Let's look at the JDBC URL syntax next.

## The JDBC URL

The JDBC URL is what is used to determine which driver implementation to use for a given `Connection`. Think of the JDBC URL (uniform resource locator) as a way to narrow down the universe of possible drivers to one specific connection. For example, suppose you need to send a package to someone. In order to narrow the universe of possible addresses down to a single unique location, you would have to identify the country, the state, the city, the street, and perhaps a house or address number on your package:

`USA:California://SanJose:FirstStreet/15`

This string indicates that the address you want is in the United States, California State, San Jose city, First Street, number 15.

JDBC URLs follow this same idea. To access Bob's Books, we might write the URL like this:

`jdbc:derby://localhost:1521/BookSellerDB`

The first part, `jdbc`, simply identifies that this is a JDBC URL (versus HTTP or something else). The second part indicates that driver vendor is `derby` driver. The third part indicates that the database is on the `localhost` of this machine (IP address `127.0.0.1`), at port `1521`, and the final part indicates that we are interested in the `BookSellerDB` database.

Just like street addresses, the reason we need this string is because JDBC was designed to work with multiple databases at once. Each of the JDBC database drivers will have a different URL, so we need to be able to pass the JDBC URL string to the `DriverManager` and ensure that the `Connection` returned was for the intended database instance.

Unfortunately, other than a requirement that the JDBC URL begin with “`jdbc`,” there is very little standard about a JDBC URL. Vendors may modify the URL to define characteristics for a particular driver implementation. The format of the JDBC URL is

`jdbc:<subprotocol>:<subname>`

In general, the subprotocol is the vendor name; for example:

`jdbc:derby`  
`jdbc:mysql`  
`jdbc:oracle`



*There are two ways to establish a connection in JDBC. The first way is using one of the few concrete classes in the `java.sql` package, `DriverManager`. The `java.sql.DriverManager` class has been a part of the JDBC implementation since the beginning, and is the easiest way to obtain a connection from a Java SE application. The alternative way is with an instance of a class that implements `javax.sql.DataSource`, introduced in JDBC 2.0.*

*Since a `DataSource` instance is typically obtained through a Java Naming and Directory Interface (JNDI) lookup, it is more often used in Java applications where there is a container that supports JNDI—for example, a Java EE application server. For the purposes of this chapter*

*(and because `DataSource` is not on the exam), we'll focus on using `DriverManager` to obtain a connection, but in the end, both ways serve to give you an instance of a `Connection` object.*

*To summarize, `DriverManager` is on the exam and `DataSource` is not.*

The subname field is where things get a bit more vendor specific. Some vendors use the subname to identify the hostname and port, followed by a database name. For example:

```
jdbc:derby://localhost:1521/MyDB
jdbc:mysql://localhost:3306/MyDB
```

Other vendors may use the subname to identify additional context information about the driver. For example:

```
jdbc:oracle:thin:@//localhost:1527/MyDB
```

In any case, it is best to consult the documentation for your specific database vendor's JDBC driver to determine the syntax of the URL.

## JDBC Driver Implementation Versions

We talked about how the `DriverManager` will scan the classpath for JAR files that contain the `META-INF/services/java.sql.Driver` file and use a classloader to load those drivers. This feature was introduced in the JDBC 4.0 specification. Prior to that, JDBC drivers were loaded manually by the application.

If you are using a JDBC driver that is an earlier version, say, a JDBC 3.0 driver, then you must explicitly load the class provided by the database vendor that implements the `java.sql.Driver` interface. Typically, the database vendor's documentation would tell you what the driver class is. For example, if our Apache Derby JDBC driver were a 3.0 driver, you would manually load the `Driver` implementation class before calling the `getConnection()` method:

```

Class.forName("org.apache.derby.jdbc.ClientDriver"); // Class loads
 // ClientDriver
try {
 Connection conn
 = DriverManager.getConnection(url, user, pwd);

```

Note that using the `Class.forName()` method is compatible with both JDBC 3.0 and JDBC 4.0 drivers. It is simply not needed when the driver supports 4.0.

Here is a quick summary of what we have discussed so far:

- Before you can start working with JDBC, creating queries and getting results, you must first establish a connection.
- In order to establish a connection, you must have a JDBC driver.
- If your JDBC driver is a JDBC 3.0 driver, then you are required to explicitly load the driver in your code using `Class.forName()` and the fully qualified path of the `Driver` implementation class.
- If your JDBC driver is a JDBC 4.0 driver, then simply include the driver (jar or zip) in the classpath.



*Although the certification exam covers up through Java SE 8, the exam developers felt they ought to include some questions about obtaining a connection using both JDBC 3.0 and JDBC 4.0 drivers. So keep in mind that for JDBC 3.0 drivers (and earlier), you are responsible for loading the class using the static `forName()` method from `java.lang.Class`.*

## CERTIFICATION OBJECTIVE

### Submit Queries and Read Results from the Database (OCP Objective 11.3)

*11.3 Submit queries and read results from the database including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections.*

In this section, we'll explore the JDBC API in much greater detail. We will start by looking at a simple example using the `Connection`, `Statement`, and `ResultSet` interfaces to pull together what we've learned so far in this chapter. Then we'll do a deep dive into `Statements` and `ResultSets`.

## All of Bob's Customers

Probably one of the most used SQL queries is `SELECT * FROM <Table name>`, which is used to print out or see all of the records in a table. Assume that we have a Java DB (Derby) database populated with data from Bob's Books. To query the database and return all of the Customers in the database, we would write something like the example shown next.

Note that to make the code listing a little shorter, going forward, we will use `out.println` instead of `System.out.println`. Just assume that means we have included a static import statement, like the one at the top of this example:

```

import static java.lang.System.*; // Static import of the
 // System class methods.
 // Now we can use just 'out'
 // instead of System.out.

String url = "jdbc:derby://localhost:1521/BookSellerDB";
String user = "bookguy";
String pwd = "$3lleR";
try {
 Connection conn =
 DriverManager.getConnection(url, user, pwd); // Get Connection
 Statement stmt = conn.createStatement(); // Create Statement
 String query = "SELECT * FROM Customer";
 ResultSet rs = stmt.executeQuery(query); // Execute Query
 while (rs.next()) { // Process Results
 out.print(rs.getInt("CustomerID") + " ");
 out.print(rs.getString("FirstName") + " ");
 out.print(rs.getString("LastName") + " ");
 out.print(rs.getString("EMail") + " ");
 out.println(rs.getString("Phone"));
 }
} catch (SQLException se) { } // Catch SQLException

```

Again, we'll dive into all of the parts of this example in greater detail, but here is what is happening:

- **Get connection** We are creating a `Connection` object instance using the information we need to access Bob's Books Database (stored on a Java DB Relational database, `BookSellerDB`, and accessed via the credentials "bookguy" with a password of "\$3lleR").

- **Create statement** We are using the connection to create a Statement object. The Statement object handles passing strings to the database as queries for the database to execute.
- **Execute query** We are executing the query string on the database and returning a ResultSet object.
- **Process results** We are iterating through the result set rows—each call to next() moves us to the next row of results.
- **Print columns** We are getting the values of the columns in the current result set row and printing them to standard out.
- **Catch SQLException** All of the JDBC API method invocations throw SQLException. A SQLException can be thrown when a method is used improperly or if the database is no longer responding. For example, a SQLException is thrown if the JDBC URL, username, or password is invalid. Or we attempted to query a table that does not exist. Or the database is no longer reachable because the network went down or the database went offline. We will look at SQLException in greater detail later in the chapter.

The output of the previous code will look something like this:

```
5000 John Smith John.Smith@comcast.net 555-340-1230
5001 Mary Johnson mary.johnson@comcast.net 555-123-4567
5002 Bob Collins bob.collins@yahoo.com 555-012-3456
5003 Rebecca Mayer rebecca.mayer@gmail.com 555-205-8212
5006 Anthony Clark anthony.clark@gmail.com 555-256-1901
5007 Judy Sousa judy.sousa@verizon.net 555-751-1207
5008 Christopher Patriquin patriquin@yahoo.com 555-316-1803
5009 Deborah Smith debsmith@comcast.net 555-256-3421
5010 Jennifer McGinn jmcmcinn@comcast.net 555-250-0918
```

We'll take a detailed look at the Statement and ResultSet interfaces and methods in the next two sections.

## Statements

Once we have successfully connected to a database, the fun can really start. From a `Connection` object, we can create an instance of a `Statement` object (or, to be precise, using the `Connection` instance we received from the `DriverManager`, we can get an instance of an object that implements the `Statement` interface). For example:

```
String url = "jdbc:derby://localhost:1521/BookSellerDB";
String user = "bookguy";
String pwd = "$3lleR";
try {
 Connection conn = DriverManager.getConnection(url, user, pwd);
 Statement stmt = conn.createStatement();
 // do stuff with SQL statements
} catch (SQLException se) { }
```

The primary purpose of a `Statement` is to execute a SQL statement using a method and return some type of result. There are several forms of `Statement` methods: those that return a result set, and those that return an integer status. The most commonly used `Statement` method performs a SQL query that returns some data, like the `SELECT` call we used earlier to fetch all the `Customer` table rows.

## Constructing and Using Statements

To start, let's look at the base `Statement`, which is used to execute a static SQL query and return a result. You'll recall that we get a `Statement` from a `Connection` and then use the `Statement` object to execute a SQL statement, like a query on the database. For example:

```
Connection conn = DriverManager.getConnection(url, user, pwd);
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM Customer");
```

Because not all SQL statements return results, the `Statement` object provides several different methods to execute SQL commands. Some SQL commands do not return a result set, but instead return an integer status. For example, SQL INSERT, UPDATE, and DELETE commands, or any of the SQL Data Definition Language (DDL) statements like CREATE TABLE, return either the number of rows affected by the query or 0.

Let's look at each of the execute methods in detail.

**public ResultSet executeQuery(String sql) throws SQLException** This is the most commonly executed `Statement` method. This method is used when we know that we want to return results—we are querying the database for one or more rows of data. For example:

```
ResultSet rs = stmt.executeQuery("SELECT * from Customer");
```

Assuming there is data in the `Customer` table, this statement should return all of the rows from the `Customer` table into a `ResultSet` object—we'll look at `ResultSet` in the next section. Notice that the method declaration includes “throws `SQLException`.” This means that this method must be called in a `try-catch` block or must be called in a method that also throws `SQLException`. Again, one reason that these methods all throw `SQLException` is that a connection to the database is likely **to a database on a network**. As with all things on the network, availability is not guaranteed, so one possible reason for `SQLException` is the lack of availability of the database itself.

**public int executeUpdate(String sql) throws SQLException** This method is used for a SQL operation that affects one or more rows and does not return results—for example, SQL INSERT, UPDATE, DELETE, and DDL queries. These statements do not return results, but do return a count of the number of rows affected by the SQL query. For example, here is an example method invocation where we want to update the `Book` table, increasing the price of every book that is currently priced less than 8.95 and is a hardcover book:

```
String q = "UPDATE Book SET UnitPrice=8.95
WHERE UnitPrice < 8.95 AND Format='Hardcover';
int numRows = stmt.executeUpdate(q);
```

When this query executes, we are expecting some number of rows will be

affected. The integer that returns is the number of rows that were updated.

Note that this `Statement` method can also be used to execute SQL queries that do not return a row count, such as `CREATE TABLE` or `DROP TABLE` and other DDL queries. For DDL queries, the return value is 0.

**public boolean execute(String sql) throws SQLException** This method is used when you are not sure what the result will be—perhaps the query will return a result set and perhaps not. This method can be used to execute a query whose type may not be known until runtime—for example, one constructed in code. The return value is true if the query resulted in a result set and false if the query resulted in an update count or no results.

However, more often, this method is used when invoking a stored procedure (using the `CallableStatement`, which we'll talk about later in the chapter). A stored procedure can return a single result set or row count, or multiple result sets and row counts, so this method was designed to handle what happens when a single database invocation produces more than one result set or row count.

You might also use this method if you wrote an application to test queries—something that reads a `String` from the command line and then runs that `String` against the database as a query. For example:

```
ResultSet rs;
int numRows;
boolean status = stmt.execute(""); // True if there is a ResultSet
if (status) { // True
 rs = stmt.getResultSet(); // Get the ResultSet
 // Process the result set...
} else { // False
 numRows = stmt.getUpdateCount(); // Get the update count
 if (numRows == -1) { // If -1, there are no results
 out.println("No results");
 }
}
```

```

 } else { // else, print the number of
 // rows affected
 out.println(numRows + " rows affected.");
 }
}

```

Because this statement may return a result set or may simply return an integer row count, there are two additional statement commands you can use to get the results or the count based on whether the execute() method returned true (there is a result set) or false (there is an update count or there was no result). The getResultSet() is used to retrieve results when the execute() method returns true, and the getUpdateCount() is used to retrieve the count when the execute() method returns false. Let's look at these methods next.



***It is generally a very bad idea to allow a user to enter a query string directly in an input field or allow a user to pass a string to construct a query directly. The reason is that if a user can construct a query or even include a freeform string into a query, he or she can use the query to return more data than you intended or alter the database table permissions.***

***For example, assume that we have a query where the user enters his e-mail address and the string the user enters is inserted directly to the query:***

```

String s = System.console().readLine("Enter your e-mail address: ");
ResultSet rs = stmt.executeQuery("SELECT * FROM Customer
 WHERE EMail=' " + s + "' ");

```

***The user of this code could enter a string like this:***

***tom@trouble.com' OR 'x='x***

***The resulting query executed by the database becomes:***

```
SELECT * FROM Customer WHERE Email='tom@trouble.com' OR 'x'='x'
```

***Because the OR statement will always return true, the result is that the query will return ALL of the customer rows, effectively the same as the query:***

```
SELECT * FROM Customer
```

***And now this user of your code has a list of the e-mail addresses of every customer in the database.***

***This type of attack is called a SQL injection attack. It is easy to prevent by carefully sanitizing any string input used in a query to the database and/or by using one of the other Statement types: PreparedStatement and CallableStatement. Despite how easy it is to prevent, it happens frequently, even to large, experienced companies like Yahoo!***

**public ResultSet getResultSet() throws SQLException** If the boolean value from the execute() method returns true, then there is a result set. To get the result set, as shown earlier, call the getResultSet() method on the Statement object. Then you can process the ResultSet object (which we will cover in the next section). This method is basically foolproof—if, in fact, there are no results, the method will return a null.

```
ResultSet rs = stmt.getResultSet();
```

**public int getUpdateCount() throws SQLException** If the boolean value from the execute() method returns false, then there is a row count, and this method will return the number of rows affected. A return value of -1 indicates that there are no results.

```
int numRows = stmt.getUpdateCount();
if (numRows == -1) {
 out.println("No results");
} else {
 out.println(numRows + " rows affected.");
}
```

[Table 12-7](#) summarizes the Statement methods we just covered.

**TABLE 12-7** Important Statement Methods

| Method (Each Throws SQLException)               | Description                                                                                                                                                                                              |
|-------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ResultSet executeQuery(String sql)</code> | Execute a SQL query and return a <code>ResultSet</code> object, i.e., SELECT commands.                                                                                                                   |
| <code>int executeUpdate(String sql)</code>      | Execute a SQL query that will only modify a number of rows, i.e. INSERT, DELETE, or UPDATE commands.                                                                                                     |
| <code>boolean execute(String sql)</code>        | Execute a SQL query that may return a result set OR modify a number of rows (or do neither). The method will return true if there is a result set or false if there may be a row count of affected rows. |
| <code>ResultSet getResultSet()</code>           | If the return value from the <code>execute()</code> method was true, you can use this method to retrieve the result set from the query.                                                                  |
| <code>int getUpdateCount()</code>               | If the return value from the <code>execute()</code> method was false, you can use this method to get the number of rows affected by the SQL command.                                                     |

## ResultSets

When a query returns a result set, an instance of a class that implements the `ResultSet` interface is returned. The `ResultSet` object represents the results of the query—all of the data in each row on a per-column basis. Again, as a reminder, *how* data in a `ResultSet` are stored is entirely up to the JDBC driver vendor. It is possible that the JDBC driver caches the entire set of results in

memory all at once, or that it uses internal buffers and gets only a few rows at a time. From your point of view as the user of the data, it really doesn't matter much. Using the methods defined in the `ResultSet` interface, you can read and manipulate the data and that's all that matters.

One important thing to keep in mind is that a `ResultSet` is a *copy* of the data from the database from the instance in time when the query was executed. Unless you are the only person using the database, you need to always assume that the underlying database table or tables that the `ResultSet` came from could be changed by some other user or application.

Because `ResultSet` is such a comprehensive part of the JDBC API, we are going to tackle it in sections. [Table 12-8](#) summarizes each section so you can reference these later.

**TABLE 12-8** `ResultSet` Sections

| Section Title                                                 | Description                                                                                                                                                                                                     |
|---------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| "Moving Forward in a ResultSet"                               | How to access each "row" of the result of a query.                                                                                                                                                              |
| "Reading Data from a ResultSet"                               | How to use <code>ResultSet</code> methods to access the individual columns of each "row" in the result set.                                                                                                     |
| "Getting Information about a ResultSet"                       | How to use a <code>ResultSetMetaData</code> object to retrieve information about the result set: the number of columns returned in the results, the names of each column, and the Java type of each column.     |
| "Printing a Report"                                           | How to use the <code>ResultSetMetaData</code> methods to print a nicely formatted set of results to the console.                                                                                                |
| "Moving Around in ResultSets"                                 | How to change the cursor type and concurrency settings on a <code>Statement</code> object to create a <code>ResultSet</code> that allows the row cursor to be positioned and allows the data to be modified.    |
| "Updating ResultSets"                                         | How to use the concurrency settings on a <code>Statement</code> object to create a <code>ResultSet</code> that allows you to update the results returned and later synchronize those results with the database. |
| "Inserting New Rows into a ResultSet"                         | How to manipulate a <code>ResultSet</code> further by deleting and inserting rows.                                                                                                                              |
| "Getting Information about a Database Using DatabaseMetaData" | How to use the <code>DatabaseMetaData</code> object to retrieve information about a database.                                                                                                                   |

## Moving Forward in a ResultSet

The best way to think of a `ResultSet` object is visually. Assume that in our BookSellerDB database we have several customers whose last name begins with the letter “C.” We could create a query to return those rows “like” this:

```
String query = "SELECT FirstName, LastName, Email from Customer
WHERE LastName LIKE 'C%';
```

The SQL operator `LIKE` treats the string that follows as a pattern to match, where the `%` indicates a wildcard. So, `Lastname LIKE 'C%`' means “any `Lastname` with a `C`, followed by any other character(s).”

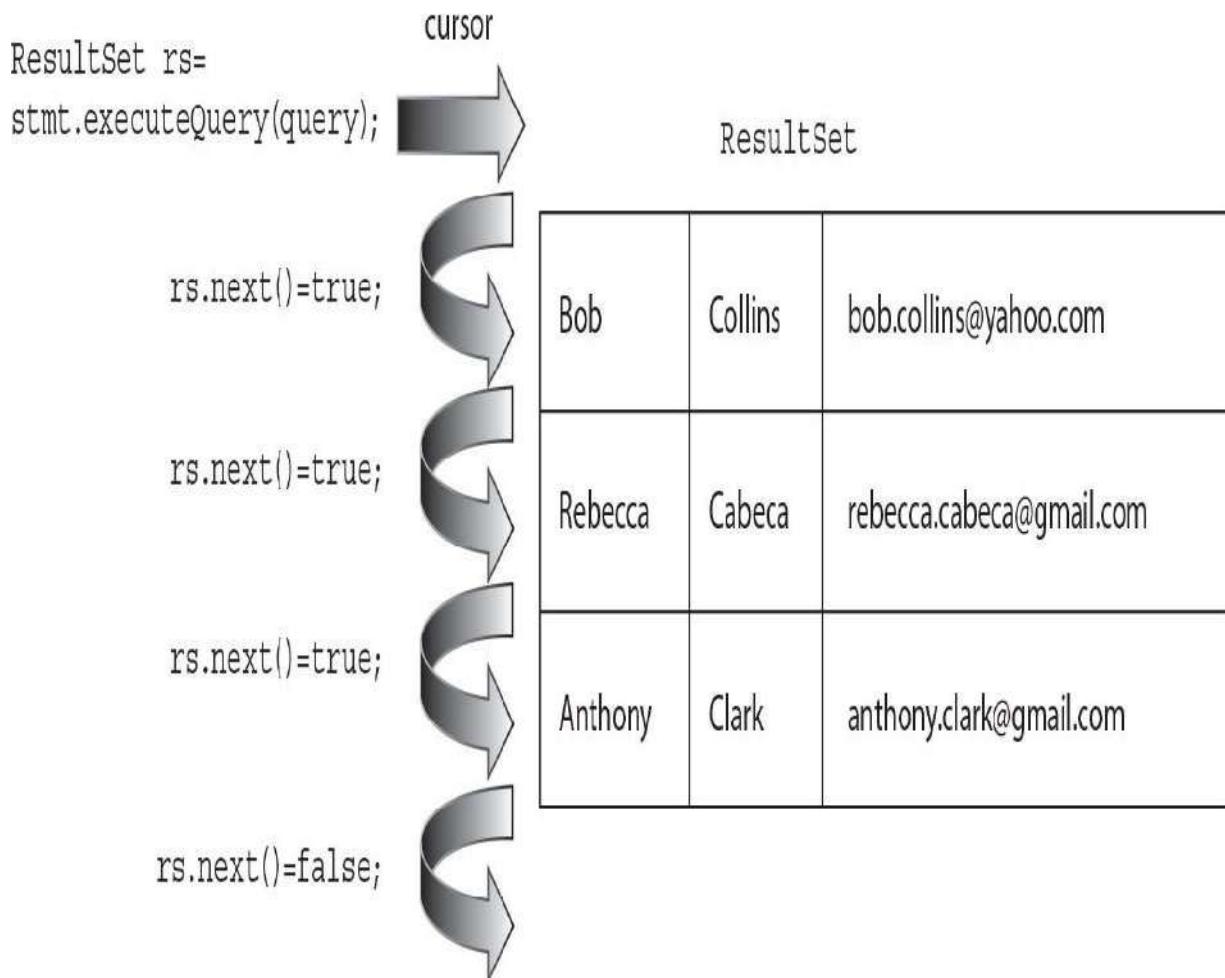
When we execute this query using the `executeQuery()` method, the `ResultSet` returned will contain the `FirstName`, `LastName`, and `Email` columns where the customer’s `LastName` starts with the capital letter “C”:

```
ResultSet rs = stmt.executeQuery (query);
```

The `ResultSet` object returned contains the data from the query as shown in [Figure 12-4](#).

**FIGURE 12-4** A `ResultSet` after the `executeQuery`

```
String query = "SELECT First _Name, Last _Name,
EMail FROM Customer WHERE Last _Name LIKE 'C%'" ;
```



Note in [Figure 12-4](#) that the `ResultSet` object maintains a cursor, or a pointer, to the current row of the results. When the `ResultSet` object is first returned from the query, the cursor is not yet pointing to a row of results—the cursor is pointing above the first row. In order to get the results of the table, you must always call the `next()` method on the `ResultSet` object to move the cursor forward to the first row of data. By default, a `ResultSet` object is read-only (the data in the rows cannot be updated), and you can only move the cursor forward. We'll look at how to change this behavior a little later on.

So the first method you will need to know for `ResultSet` is the `next()` method.

**public boolean next()** The `next()` method moves the cursor forward one row

and returns true if the cursor now points to a row of data in the `ResultSet`. If the cursor points beyond the last row of data as a result of the `next()` method (or if the `ResultSet` contains no rows), the return value is false.

So in order to read the three rows of data in the table shown in [Figure 12-4](#), we need to call the `next()` method, read the row of data, and then call `next()` again twice more. When the `next()` method is invoked the fourth time, the method will return false. The easiest way to read all of the rows from first to last is in a `while` loop:

```
String query = "SELECT FirstName, LastName, EMail FROM Customer
 WHERE LastName LIKE 'C%'";

ResultSet rs = stmt.executeQuery(query);
while (rs.next()) { // Move the cursor from the current position
 // to the next row of data - return true if the
 // next row is valid data and false if the
 // cursor has moved past the last row
 // ...
}
```



***Because the cursor is such a fundamental concept in JDBC, the exam will test you on the status of the cursor in a ResultSet. As long as you keep in mind that you must call the next() method before processing even one row of data in a ResultSet, then you'll be fine. Maybe you could use a memory device like this one: "When getting results, don't vex, always call next!" Okay, maybe not.***

## Reading Data from a `ResultSet`

Moving the cursor forward through the `ResultSet` is just the start of reading data from the results of the query. Let's look at the two ways to get the data from each row in a result set.

When a `ResultSet` is returned and you have dutifully called `next()` to move the cursor to the first actual row of data, you can now read the data in each column of the current row. As illustrated in [Figure 12-4](#), a result set from a database query is like a table or a spreadsheet. Each row contains (typically) one or more columns, and the data in each column is one of the SQL data types. In order to bring the data from each column into your Java application, you must use a `ResultSet` method to retrieve each of the SQL column values into an appropriate Java type. So SQL `INTEGER`, for example, can be read as a Java `int` primitive, SQL `VARCHAR` can be read as a Java `String`, SQL `DATE` can be read as a `java.sql.Date` object, and so on. `ResultSet` defines several other types as well, but whether or not the database or the driver supports all of the types defined by the specification depends on the database vendor. For the exam, we recommend you focus on the most common SQL data types and the `ResultSet` methods shown in [Table 12-9](#).

**TABLE 12-9** SQL Types and JDBC Types

| <b>SQL Type</b>               | <b>Java Type</b>   | <b>ResultSet get methods</b>                                     |
|-------------------------------|--------------------|------------------------------------------------------------------|
| BOOLEAN                       | boolean            | getBoolean(String columnName)<br>getBoolean(int columnIndex)     |
| INTEGER                       | int                | getInt(String columnName)<br>getInt(int columnIndex)             |
| DOUBLE, FLOAT                 | double             | getDouble(String columnName)<br>getDouble(int columnIndex)       |
| REAL                          | float              | getFloat(String columnName)<br>getFloat(int columnIndex)         |
| BIGINT                        | long               | getLong(String columnName)<br>getLong(int columnIndex)           |
| CHAR, VARCHAR,<br>LONGVARCHAR | String             | getString(String columnName)<br>getString(int columnIndex)       |
| DATE                          | java.sql.Date      | getDate(String columnName)<br>getDate(int columnIndex)           |
| TIME                          | java.sql.Time      | getTime(String columnName)<br>getTime(int columnIndex)           |
| TIMESTAMP                     | java.sql.Timestamp | getTimestamp(String columnName)<br>getTimestamp(int columnIndex) |
| Any of the above              | java.lang.Object   | getObject(String columnName)<br>getObject(int columnIndex)       |



*SQL has been around for a long time. The first formalized American National Standards Institute (ANSI)-approved version was adopted in 1986 (SQL-86). The next major revision was in 1992, SQL-92, which is widely considered the “base” release for every database. SQL-92 defined a number of new data types, including DATE, TIME, TIMESTAMP, BIT, and VARCHAR strings. SQL-92 has multiple levels; each level adds a bit more functionality to the previous level. JDBC drivers recognize three ANSI SQL-92 levels: Entry, Intermediate, and Full.*

*SQL-1999, also known as SQL-3, added LARGE OBJECT types, including BINARY LARGE OBJECT (BLOB) and CHARACTER LARGE OBJECT (CLOB). SQL-1999 also introduced the BOOLEAN type and a composite type, ARRAY and ROW, to store collections directly into the database. In addition, SQL-1999 added a number of features to SQL, including triggers, regular expressions, and procedural and flow control.*

*SQL-2003 introduced XML to the database, and importantly, added columns with auto-generated values, including columns that support identity, like the primary key and foreign key columns. Believe it or not, other standards have been proposed, including SQL-2006, SQL-2008, and SQL-2011.*

*The reason this matters is because the JDBC specification has attempted to be consistent with features from the most widely adopted specification at the time. Thus, JDBC 3.0 supports SQL-92 and a part of the SQL-1999 specification, and JDBC 4.0 supports parts of the SQL-2003 specification. In this chapter, we’ll try to stick to the most widely used SQL-92 features and the most commonly supported SQL-1999 features that JDBC also supports.*

One way to read the column data is by using the names of the columns themselves as string values. For example, using the column names from Bob’s Book table ([Table 12-4](#)), in these ResultSet methods, the String name of the column from the Book table is passed to the method to read the column data type:

```

String query = "SELECT Title, PubDate, Price FROM Book";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
 String title = rs.getString("Title"); // Read the data in the
 // column named "Title"
 // into a String
 Date PubDate = rs.getDate("PubDate"); // Read the data in the
 // "PubDate" column into
 // a java.sql.Date object
 float price = rs.getFloat("Price"); // Read the data in the
 // column "Price"
 // into a float
 //
}

```

Note that although here the column names were retrieved from the `ResultSet` row in the order they were requested in the SQL query, they could have been processed in any order.

`ResultSet` also provides an overloaded method that takes an integer index value for each of the SQL types. This value is the integer position of the column in the result set, numbered from 1 to the number of columns returned. So we could write the same statements earlier like this:

```

String title = rs.getString(1); // Title is first column
Date PubDate = rs.getDate(2); // PubDate is second column
float price = rs.getFloat(3); // Price is third column

```

Using the positional methods shown earlier, the order of the column in the `ResultSet` does matter. In our query, Title is in position 1, PubDate is in position 2, and Price is in position 3.

**Remember: Column indexes start with 1.**

***It is important to keep in mind that when you are accessing columns using integer index values, the column indexes always start with 1, not 0 as in traditional arrays. If you attempt to access a column with an index of less than 1 or greater than the number of columns returned, a SQLException will be thrown. You can get the number of columns returned in a ResultSet through the result set's metadata object. See the section on ResultSetMetaData to learn more.***

  
**on the  
job**

***What the database stores as a type, the SQL type, and what JDBC returns as a type are often two different things. It is important to understand that the JDBC specification provides a set of standard mappings—the best match between what the database provides as a type and the Java type a programmer should use with that type. Rather than repeating what is in the specification, we encourage you to look at Appendix B of the JDBC (JSR-221) specification.***

The most commonly used `ResultSet` get methods are listed next. Let's look at these methods in detail.

**public boolean getBoolean(String columnLabel)** This method retrieves the value of the named column in the `ResultSet` as a Java boolean. Boolean values are rarely returned in SQL queries, and some databases may not support a SQL BOOLEAN type, so check with your database vendor. In this contrived example here, we are returning employment status:

```
if (rs.getBoolean("CURR_EMPLOYEE")) {
 // Now process the remaining columns
}
```

**public double getDouble(String columnLabel)** This method retrieves the value of the column as a Java double. This method is recommended for returning the value stored in the database as SQL DOUBLE and SQL FLOAT types.

```
double cartTotal = rs.getDouble("CartTotal");
```

**public int getInt(String columnLabel)** This method retrieves the value of the column as a Java int. Integers are often a good choice for primary keys. This method is recommended for returning values stored in the database as SQL INTEGER types.

```
int authorID = rs.getInt("AuthorID");
```

**public float getFloat(String columnLabel)** This method retrieves the value of the column as a Java float. It is recommended for SQL REAL types.

```
float price = rs.getFloat("UnitPrice");
```

**public long getLong(String columnLabel)** This method retrieves the value of the column as a Java long. It is recommended for SQL BIGINT types.

```
long socialSecurityNumber = rs.get("SocSecNum");
```

**public java.sql.Date getDate(String columnLabel)** This method retrieves the value of the column as a Java Date object. Note that java.sql.Date extends java.util.Date. One difference between the two is that the `toString()` method of java.sql.Date returns a date string in the form: “yyyy mm dd.” This method is recommended for SQL DATE types.

```
java.sql.Date pubDate = rs.getDate("PubDate");
```

**public java.lang.String getString(String columnLabel)** This method retrieves the value of the column as a Java String object. It is good for reading SQL columns with CHAR, VARCHAR, and LONGVARCHAR types.

```
String lastName = rs.getString("LastName");
```

**public java.sql.Time getTime(String columnLabel)** This method retrieves the value of the column as a Java Time object. Like java.sql.Date, this class extends java.util.Date, and its `toString()` method returns a time string in the

form: “hh:mm:ss.” TIME is the SQL type that this method is designed to read.

```
java.sql.Time time = rs.getTime("FinishTime");
```

**public java.sql.Timestamp getTimestamp(String columnLabel)** This method retrieves the value of the column as a Timestamp object. Its `toString()` method formats the result in the form: yyyy-mm-dd hh:mm:ss.fffffffff, where ffffffff is nanoseconds. This method is recommended for reading SQL TIMESTAMP types.

```
java.sql.Timestamp timestamp = rs.getTimestamp("ClockInTime");
```

**public java.lang.Object getObject(String columnLabel)** This method retrieves the value of the column as a Java Object. It can be used as a general-purpose method for reading data in a column. This method works by reading the value returned as the appropriate Java wrapper class for the type and returning that as a Java Object object. So, for example, reading an integer (SQL INTEGER type) using this method returns an object that is a `java.lang.Integer` type. We can use `instanceof` to check for an Integer and get the int value:

```
Object o = rs.getObject("AuthorID");
if (o instanceof java.lang.Integer) {
 int id = ((Integer)o).intValue();
}
```

[Table 12-9](#) lists the most commonly used methods to retrieve specific data from a `ResultSet`. For the complete and exhaustive set of `ResultSet` get methods, see the Java documentation for `java.sql.ResultSet`.



*The exam is not going to test your knowledge of all of the `ResultSet` get and set methods for SQL types. For the exam, just remember the basic Java types: `String` and `int`. Each `ResultSet` getter method is named by its closest Java type, so, for example, to read a database column that holds an integer into a Java `int` type, you invoke the `getInt()` method with either*

***the string column or the column index of the column you wish to read.***

## Getting Information about a ResultSet

When you write a query using a string, as we have in the examples so far, you know the name and type of the columns returned. However, what happens when you want to allow your users to dynamically construct the query? You may not always know in advance how many columns are returned and the type and name of the columns returned.

Fortunately, the `ResultSetMetaData` class was designed to provide just that information. Using `ResultSetMetaData`, you can get important information about the results returned from the query, including the number of columns, the table name, the column name, and the column class name—the Java class that is used to represent this column when the column is returned as an `Object`. Here is a simple example, and then we'll look at these methods in more detail:

```
String query = "SELECT AuthorID FROM Author";
ResultSet rs = stmt.executeQuery(query);
ResultSetMetaData rsmd = rs.getMetaData();
rs.next();
int colCount = rsmd.getColumnCount(); // How many columns in this
 // ResultSet?
out.println("Column Count: " + colCount);
for (int i = 1; i <= colCount; i++) {
 out.println("Table Name: " + rsmd.getTableName(i));
 out.println("Column Name: " + rsmd.getColumnName(i));
 out.println("Column Size: " + rsmd.getColumnDisplaySize(i));
}
```

Running this code using the BookSeller database (Bob's Books) produces the following output:

```
Column Count: 1
Table Name: AUTHOR
Column Name: AUTHORID
Column Size: 11
```

ResultSetMetaData is often used to generate reports, so here are the most commonly used methods. For more information and more methods, check out the JavaDocs.

**public int getColumnCount() throws SQLException** This method is probably the most used ResultSetMetaData method. It returns the integer count of the number of columns returned by the query. With this method, you can iterate through the columns to get information about each column.

```
try {
 conn = DriverManager.getConnection(...);
 stmt = conn.createStatement();
 String query = "SELECT * FROM Author";

 ResultSet rs = stmt.executeQuery(query);
 ResultSetMetaData rsmd = rs.getMetaData(); // Get the meta data
 // for this ResultSet
 int columnCount = rsmd.getColumnCount(); // Get the number
 // of columns in this
 // ResultSet
 ...
} catch (SQLException se) { }
```

The value of columnCount for the Author table is 3. We can use this value to iterate through the columns using the methods illustrated next.

**public String getColumnName(int column) throws SQLException** This method returns the String name of this column. Using the columnCount, we can create an output of the data from the database in a report-like format. For

example:

```
String colData;
ResultSet rs = stmt.executeQuery(query);
ResultSetMetaData rsmd = rs.getMetaData();
int cols = rsmd.getColumnCount();
for (int i = 1; i <= cols; i++) {
 out.print(rsmd.getColumnName(i) + " "); // Print each column name
}
out.println();
while (rs.next()) {
 for (int i = 1; i <= cols; i++) {
 if (rs.getObject(i) != null) {
 colData = rs.getObject(i).toString(); // Get the String value
 // of the column object
 } else {
 colData = "NULL"; // or NULL for a null
 }
 out.print(colData); // Print the column data
 }
 out.println();
}
```

This example is somewhat rudimentary, as we probably need to do some better formatting on the data, but it will produce a table of output:

| AUTHORID | FIRSTNAME | LASTNAME |
|----------|-----------|----------|
| 1000     | Rick      | Riordan  |
| 1001     | Nancy     | Farmer   |
| 1002     | Ally      | Condie   |
| 1003     | Cressida  | Cowell   |
| 1004     | Lauren    | St. John |
| 1005     | Eoin      | Colfer   |

...

**public String getTableName(int column) throws SQLException** The method returns the String name of the table that this column belongs to. This method is useful when the query is a join of two or more tables and we need to know which table a column came from. For example, suppose that we want to get a list of books by author's last name:

```
String query = "SELECT Author.LastName, Book.Title
 FROM Author, Book, Books_By_Author
 WHERE Author.AuthorID = Books_By_Author.AuthorID
 AND Book.isbn = Books_By_Author.isbn"
```

With a query like this, we might want to know which table the column data came from:

```
ResultSetMetaData rsmd = rs.getMetaData();
int cols = rsmd.getColumnCount();
for (int i = 1; i <= cols; i++) {
 out.print(rsmd.getTableName(i) + ":" +
 rsmd.getColumnName(i) + " ");
}
```

This code will print the name of the table, a colon, and the column name. The output might look something like this:

AUTHOR : LASTNAME BOOK : TITLE

**public int getColumnDisplaySize(int column) throws SQLException** This method returns an integer of the size of the column. This information is useful for determining the maximum number of characters a column can hold (if it is a VARCHAR type) and the spacing that is required between columns for a report.

## Printing a Report

To make a prettier report than the one in the `getColumnName` method earlier, for example, we could use the display size to pad the column name and data with spaces. What we want is a table with spaces between the columns and headings that looks something like this when we query the Author table:

| AUTHORID | FIRSTNAME | LASTNAME |
|----------|-----------|----------|
| 1000     | Rick      | Riordan  |
| 1001     | Nancy     | Farmer   |
| 1002     | Ally      | Condie   |
| 1003     | Cressida  | Cowell   |
| 1004     | Lauren    | St. John |
| 1005     | Eoin      | Colfer   |
| ...      |           |          |

Using the methods we have discussed so far, here is code that produces a pretty report from a query:

```
ResultSet rs = stmt.executeQuery(query);
ResultSetMetaData rsmd = rs.getMetaData();
int cols = rsmd.getColumnCount();
String col, colData;
for (int i = 1; i <= cols; i++) {
 col = leftJustify(rsmd.getColumnName(i), // Left justify
 rsmd.getColumnDisplaySize(i)); // column name
 out.print(col); // padded with
} // size spaces
out.println(); // Print a linefeed
while (rs.next()) {
 for (int i = 1; i <= cols; i++) {
 if (rs.getObject(i) != null) {
 colData = rs.getObject(i).toString(); // Get the data in the
 // column as a String
 } else {
 colData = "NULL"; // If the column is null
 // use "NULL"
 }
 col = leftJustify(colData,
 rsmd.getColumnDisplaySize(i)));
 out.print(col);
 }
 out.println();
}
```

A couple of things to note about the example code: first, the `leftJustify` method, which takes a string to print left-justified and an integer for the total number of characters in the string. The difference between the actual string length and the integer value will be filled with spaces. This method uses the `String format()` method and the " - " (dash) flag to return a `String` that is left-justified with spaces. The `%1$` part indicates the flag should be applied to the first argument. What we are building is a format string dynamically. If the column display size is 20, the format string will be `%1$-20s`, which says “print the argument passed (the first argument) on the left with a width of 20 and use a string conversion.”

Note that if the length of the string passed in and the integer field length (`n`) are the same, we add one space to the length to make it look pretty:

```
public static String leftJustify(String s, int n) {
 if (s.length() <= n) n++; // Add an extra space if the length of
 // the String s is less than or equal to
 // the length of the column n
 return String.format("%1$-" + n + "s", s); // Pad to the right of
 // the String by n
 // spaces
}
```

Second, databases can store NULL values. If the value of a column is NULL, the object returned in the `rs.getObject()` method is a Java null. So we have to test for null to avoid getting a null pointer exception when we execute the `toString()` method.

Notice that we don’t have to use the `next()` method before reading the `ResultSetMetaData`—we can do that at any time after obtaining a valid result set. Running this code and passing it a query like “SELECT \* FROM Author” returns a neatly printed set of authors:

| AUTHORID | FIRSTNAME | LASTNAME |
|----------|-----------|----------|
| 1000     | Rick      | Riordan  |
| 1001     | Nancy     | Farmer   |
| 1002     | Ally      | Condie   |
| 1003     | Cressida  | Cowell   |
| 1004     | Lauren    | St. John |
| 1005     | Eoin      | Colfer   |
| ...      |           |          |

## Moving Around in ResultSets

So far, for all the result sets we looked at, we simply moved the cursor forward by calling `next()`. The default characteristics of a `Statement` are cursors that only move forward and result sets that do not support changes. The `ResultSet` interface actually defines these characteristics as static int variables:

`TYPE_FORWARD_ONLY` and `CONCUR_READ_ONLY`. However, the JDBC specification defines additional static int types (shown next) that allow a developer to move the cursor forward, backward, and to a specific position in the result set. In addition, the result set can be modified while open and the changes written to the database. Note that support for cursor movement and updatable result sets is not a requirement on a driver, but most drivers provide this capability. In order to create a result set that uses positionable cursors and/or supports updates, you must create a `Statement` with the appropriate scroll type and concurrency setting, and then use that `Statement` to create the `ResultSet` object.

The ability to move the cursor to a particular position is the key to being able to determine how many rows are returned from a result set—something we will look at shortly. The ability to modify an open result set may seem odd, particularly if you are a seasoned database developer. After all, isn't that what a SQL `UPDATE` command is for?

Consider a situation in which you want to perform a series of calculations using the data from the result set rows, then write a change to each row based on some criteria, and finally write the data back to the database. For example, imagine a database table that contains customer data, including the date they joined as a customer, their purchase history, and the total number of orders in the last two months. After reading this data into a result set, you could iterate over

each customer record and modify it based on business rules: set their minimum discount higher if they have been a customer for more than a year with at least one purchase per year or set their preferred credit status if they have been purchasing more than \$100 per month. With an updatable result set, you can modify several customer rows, each in a different way, and commit the rows to the database without having to write a complex SQL query or a set of SQL queries—you simply commit the updates on the open result set.

Let's look at how to modify a result set in more detail. There are three `ResultSet` cursor types:

- **TYPE\_FORWARD\_ONLY** The default value for a `ResultSet`—the cursor moves forward only through a set of results.
- **TYPE\_SCROLL\_INSENSITIVE** A cursor position can be moved in the result forward or backward, or positioned to a particular cursor location. Any changes made to the underlying data—the database itself—are not reflected in the result set. In other words, the result set does not have to “keep state” with the database. This type is generally supported by databases.
- **TYPE\_SCROLL\_SENSITIVE** A cursor can be moved in the results forward or backward, or positioned to a particular cursor location. Any changes made to the underlying data are reflected in the open result set. As you can imagine, this is difficult to implement and is, therefore, not implemented in a database or JDBC driver very often.

JDBC provides two options for data concurrency with a result set:

- **CONCUR\_READ\_ONLY** This is the default value for result set concurrency. Any open result set is read-only and cannot be modified or changed.
- **CONCUR\_UPDATABLE** A result set can be modified through the `ResultSet` methods while the result set is open.

Because a database and JDBC driver are not required to support cursor movement and concurrent updates, the JDBC provides methods to query the database and driver using the `DatabaseMetaData` object to determine if your driver supports these capabilities. For example:

```
Connection conn = DriverManager.getConnection(...);
DatabaseMetaData dbmd = conn.getMetaData();
if (dbmd.supportsResultSetType(ResultSet.TYPE_FORWARD_ONLY)) {
 out.print("Supports TYPE_FORWARD_ONLY");
 if (dbmd.supportsResultSetConcurrency(
 ResultSet.TYPE_FORWARD_ONLY,
 ResultSet.CONCUR_UPDATABLE)) {
 out.println(" and supports CONCUR_UPDATABLE");
 }
}

if (dbmd.supportsResultSetType(ResultSet.TYPE_SCROLL_INSENSITIVE)) {
 out.print("Supports TYPE_SCROLL_INSENSITIVE");
 if (dbmd.supportsResultSetConcurrency(
 ResultSet.TYPE_SCROLL_INSENSITIVE,
 ResultSet.CONCUR_UPDATABLE)) {
 out.println(" and supports CONCUR_UPDATABLE");
 }
}

if (dbmd.supportsResultSetType(ResultSet.TYPE_SCROLL_SENSITIVE)) {
 out.print("Supports TYPE_SCROLL_SENSITIVE");
 if (dbmd.supportsResultSetConcurrency(
 ResultSet.TYPE_SCROLL_SENSITIVE,
 ResultSet.CONCUR_UPDATABLE)) {
 out.println("Supports CONCUR_UPDATABLE");
 }
}
```

Running this code on the Java DB (Derby) database, these are the results:

Supports TYPE\_FORWARD\_ONLY and supports CONCUR\_UPDATABLE

Supports TYPE\_SCROLL\_INSENSITIVE and supports CONCUR\_UPDATABLE

In order to create a `ResultSet` with `TYPE_SCROLL_INSENSITIVE` and `CONCUR_UPDATABLE`, the `Statement` used to create the `ResultSet` must be created (from the `Connection`) with the cursor type and concurrency you want. You can determine what cursor type and concurrency the `Statement` was created with, but once created, you can't change the cursor type or concurrency of an existing `Statement` object. Also, note that just because you set a cursor type or concurrency setting, that doesn't mean you will get those settings. As you will see in the section on exceptions, the driver can determine that the database doesn't support one or both of the settings you chose and it will throw a warning and (silently) revert to its default settings, if they are not supported. You will see how to detect these JDBC warnings in the section on exceptions and warnings.

```
Connection conn = DriverManager.getConnection(...);
Statement stmt =
 conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
 ResultSet.CONCUR_UPDATABLE);
```

Besides being able to use a `ResultSet` object to update results, which we'll look at next, being able to manipulate the cursor provides a side benefit—we can use the cursor to determine the number of rows returned in a query. Although it would seem like there ought to be a method in `ResultSet` or `ResultSetMetaData` to do this, this method does not exist.

In general, you should not need to know how many rows are returned, but during debugging, you may want to diagnose your queries with a stand-alone database and use cursor movement to read the number of rows returned.

Something like this would work:

```
ResultSet rs = stmt.executeQuery(query); // Get a ResultSet
if (rs.last()) { // Move the very last row
 int rowCount = rs.getRow(); // Get row number (the count)
 rs.beforeFirst(); // Move to before the 1st row
}
```

Of course, you may also want to have a more sophisticated method that preserves the current cursor position and returns the cursor to that position, regardless of when the method was called. Before we look at that code, let's look at the other cursor movement methods and test methods (besides next) in `ResultSet`. As a quick summary, [Table 12-10](#) lists the methods you use to change the cursor position in a `ResultSet`.

**TABLE 12-10** `ResultSet` Cursor Positioning Methods

| Method                                 | Effect on the Cursor and Return Value                                                                                                                                                                                                                                                                                                        |
|----------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>boolean next()</code>            | Moves the cursor to the next row in the <code>ResultSet</code> . Returns <code>false</code> if the cursor is positioned beyond the last row.                                                                                                                                                                                                 |
| <code>boolean previous()</code>        | Moves the cursor backward one row. Returns <code>false</code> if the cursor is positioned before the first row.                                                                                                                                                                                                                              |
| <code>boolean absolute(int row)</code> | Moves the cursor to an absolute position in the <code>ResultSet</code> . Rows are numbered from 1. Moving to row 0 moves the cursor to before the first row. Moving to negative row numbers starts from the last row and works backward. Returns <code>false</code> if the cursor is positioned beyond the last row or before the first row. |
| <code>boolean relative(int row)</code> | Moves the cursor to a position relative to the current position. Invoking <code>relative(1)</code> moves forward one row; invoking <code>relative(-1)</code> moves backward one row. Returns <code>false</code> if the cursor is positioned beyond the last row or before the first row.                                                     |
| <code>boolean first()</code>           | Moves the cursor to the first row in the <code>ResultSet</code> . Returns <code>false</code> if there are no rows in the <code>ResultSet</code> (empty result set).                                                                                                                                                                          |
| <code>boolean last()</code>            | Moves the cursor to the last row in the <code>ResultSet</code> . Returns <code>false</code> if there are no rows in the <code>ResultSet</code> (empty result set).                                                                                                                                                                           |
| <code>void beforeFirst()</code>        | Moves the cursor to before the first row in the <code>ResultSet</code> .                                                                                                                                                                                                                                                                     |
| <code>void afterLast()</code>          | Moves the cursor to after the last row in the <code>ResultSet</code> .                                                                                                                                                                                                                                                                       |

Let's look at each of these methods in more detail.

**public boolean absolute(int row) throws SQLException** This method positions the cursor to an absolute row number. The contrasting method is relative. Passing 0 as the row argument positions the cursor to before the first row. Passing a negative value, like -1, positions the cursor to the position after the last row minus one—in other words, the last row. If you attempt to position the cursor beyond the last row, say at position 22 in a 19-row result set, the cursor will be positioned beyond the last row, the implications of which we'll discuss next. [Figure 12-5](#) illustrates how invocations of `absolute()` position the cursor.

### FIGURE 12-5

Absolute cursor positioning

```
String query = "SELECT * FROM Author";
```

| cursor | ResultSet |             |          |
|--------|-----------|-------------|----------|
| 1      | 1000      | Rick        | Riordan  |
| 2      | 1001      | Nancy       | Farmer   |
| 3      | 1002      | Ally        | Condie   |
| 4      | 1003      | Cressida    | Cowell   |
| 5      | 1004      | Lauren      | St. John |
| 6      | 1005      | Eoin        | Colfer   |
| 7      | 1006      | Esther      | Freisner |
| 8      | 1007      | Chris       | D'lacey  |
| 9      | 1008      | Christopher | Paolini  |
| 10     | 1009      | Kathryn     | Lasky    |
| 11     | 1010      | Nancy       | Star     |

The `absolute()` method returns `true` if the cursor was successfully positioned within the `ResultSet` and `false` if the cursor ended up before the first or after the last row. For example, suppose you wanted to process only every other row:

```

ResultSet rs = stmt.executeQuery(query);
for (int i = 1; ; i += 2) {
 if (rs.absolute(i)) { // The absolute method moves to the row
 // passed as the integer value and returns
 // true if the move was successful
 // ... process the odd row
 } else {
 break;
 }
}

```

**public int getRow() throws SQLException** This method returns the current row position as a positive integer (1 for the first row, 2 for the second, and so on) or 0 if there is no current row—the cursor is either before the first row or after the last row. This is the only method of this set of cursor methods that is optionally supported for TYPE\_FORWARD\_ONLY ResultSets.

**public boolean relative(int rows) throws SQLException** The relative() method is the cousin to absolute. Get it, cousin? Okay, anyway, relative() will position the cursor either before or after the current position of the number of rows passed in to the method. So if the cursor is on row 15 of a 30-row ResultSet, calling relative(2) will position the cursor to row 17, and then calling relative(-5) positions the cursor to row 12. [Figure 12-6](#) shows how the cursor is moved based on calls to absolute() and relative().

## FIGURE 12-6

Relative cursor positioning (Circled numbers indicate order of invocation.)

```
String query = "SELECT * FROM Author";
```

|                      | cursor | ResultSet                |
|----------------------|--------|--------------------------|
| ① rs.absolute(2); ➔  | 1      | 1000 Rick Riordan        |
|                      | 2      | 1001 Nancy Farmer        |
|                      | 3      | 1002 Ally Condie         |
| ③ rs.relative(-3); ➔ | 4      | 1003 Cressida Cowell     |
|                      | 5      | 1004 Lauren St. John     |
|                      | 6      | 1005 Eoin Colfer         |
| ② rs.relative(5); ➔  | 7      | 1006 Esther Freisner     |
|                      | 8      | 1007 Chris D'lacey       |
|                      | 9      | 1008 Christopher Paolini |
|                      | 10     | 1009 Kathryn Lasky       |
|                      | 11     | 1010 Nancy Star          |

Like absolute positioning, attempting to position the cursor beyond the last row or before the first row simply results in the cursor being after the last row or before the first row, respectively, and the method returns false. Also, calling relative with an argument of 0 does exactly what you might expect—the cursor

remains where it is. Why would you use relative? Let's assume you are displaying a fairly long database table on a web page using an HTML table. You might want to allow your user to be able to page forward or backward relative to the currently selected row—maybe something like this:

```
public boolean getNextPageOfData (ResultSet rs, int pageSize) throws
SQLException{
 return rs.relative(pageSize);
}
```

**public boolean previous() throws SQLException** The `previous()` method works exactly the same as the `next()` method, only it backs up through the `ResultSet`. Using this method with the `afterLast()` method described next, you can move through a `ResultSet` in reverse order (from last row to first).

**public void afterLast() throws SQLException** This method positions the cursor after the last row. Using this method and then the `previous()` method, you can iterate through a `ResultSet` in reverse. For example:

```
public void showFlippedResultSet(ResultSet rs) throws SQLException {
 rs.afterLast(); // Position the cursor after the last row
 while (rs.previous()) { // Back up through the ResultSet
 // process the result set
 }
}
```

Just like `next()`, when `previous()` backs up all the way to before the first row, the method returns `false`.

**public void beforeFirst() throws SQLException** This method will return the cursor to the position it held when the `ResultSet` was first created and returned by a `Statement` object.

```
rs.beforeFirst(); // Position the cursor before the first row
```

**public boolean first() throws SQLException** The `first()` method positions the cursor on the first row. It is the equivalent of calling `absolute(1)`. This method returns `true` if the cursor was moved to a valid row and `false` if the `ResultSet` has no rows.

```
if (!rs.first()) {
 out.println("No rows in this result set");
}
```

**public boolean last() throws SQLException** The `last()` method positions the cursor on the last row. This method is the equivalent of calling `absolute(-1)`. This method returns `true` if the cursor was moved to a valid row and `false` if the `ResultSet` has no rows.

```
if (!rs.last()) {
 out.println("No rows in this result set");
}
```

A couple of notes on the exceptions thrown by all of these methods:

- A `SQLException` will be thrown by these methods if the type of the `ResultSet` is `TYPE_FORWARD_ONLY`, if the `ResultSet` is closed (we will look at how a result set is closed in an upcoming section), or if a database error occurs.
- A `SQLFeatureNotSupportedException` will be thrown by these methods if the JDBC driver does not support the method. This exception is a subclass of `SQLException`.
- Most of these methods have no effect if the `ResultSet` has no rows—for example, a `ResultSet` returned by a query that returned no rows.

The following methods return a boolean to allow you to “test” the current cursor position without moving the cursor. Note that these are not on the exam but are provided to you for completeness:

- `isBeforeFirst()` True if the cursor is positioned before the first row
- `isAfterLast()` True if the cursor is positioned after the last row

- **isFirst()** True if the cursor is on the first row
- **isLast()** True if the cursor is on the last row

So now that we have looked at the cursor positioning methods, let's revisit the code to calculate the row count. We will create a general-purpose method to allow the row count to be calculated at any time and at any current cursor position. Here is the code:

```
public static int getRowCount(ResultSet rs) throws SQLException {
 int rowCount = -1;
 int currRow = 0;

 if (rs != null) { // make sure the ResultSet is not null
 currRow = rs.getRow(); // Save the current row position:
 // zero indicates that there is no
 // current row position - could be
 // beforeFirst or afterLast
 if (rs.isAfterLast()) { // afterLast, so set the currRow negative
 currRow = -1;
 }
 if (rs.last()) { // move to the last row and get the position
 // if this method returns false, there are no
 // results
 rowCount = rs.getRow(); // Get the row count
 // Return the cursor to the position it
 // was in before the method was called.
 if (currRow == -1) { // if the currRow is negative, the cursor
 // position was after the last row, so
 // return the cursor to the last row
 rs.afterLast();
 } else if (currRow == 0) { // else if the cursor is zero, move
 // the cursor to before the first row
 rs.beforeFirst();
 } else { // else return the cursor to its last position
 rs.absolute(currRow);
 }
 }
 }
 return rowCount;
}
```

Looking through the code, you notice that we took special care to preserve the current position of the cursor in the `ResultSet`. We called `getRow()` to get the current position, and if the value returned was 0, the current position of the `ResultSet` could be either before the first row or after the last row, so we used the `isAfterLast()` method to determine where the cursor was. If the cursor was after the last row, then we stored a -1 in the `currRow` integer.

We then moved the cursor to the last position in the `ResultSet`, and if that move was successful, we get the current position and save it as the `rowCount` (the last row and, therefore, the count of rows in the `ResultSet`). Finally, we use the value of `currRow` to determine where to return the cursor. If the value of the cursor is -1, we need to position the cursor after the last row. Otherwise, we simply use `absolute()` to return the cursor to the appropriate position in the `ResultSet`.

While this may seem like several extra steps, we will look at why preserving the cursor can be important when we look at updating `ResultSets` next.

## Updating ResultSets

**Please note that you might not get any questions on the real exam for this section and the subsections that follow.**

If you have casually used JDBC, or are new to JDBC, you may be surprised to know that a `ResultSet` object can do more than just provide the results of a query to your application. Besides just returning the results of a query, a `ResultSet` object may be used to modify the contents of a database table, including update existing rows, delete existing rows, and add new rows.

In a traditional SQL application, you might perform the following SQL queries to raise the price of all of the hardcover books in inventory that are currently 10.95 to 11.95 in price:

```
UPDATE Book SET UnitPrice = 11.95 WHERE UnitPrice = 10.95
AND Format = 'Hardcover'
```

Hopefully, by now you feel comfortable creating a statement to perform this query using a SQL UPDATE:

```
// We have a connection and we are in a try-catch block...
Statement stmt = conn.createStatement();
String query = "UPDATE Book SET UnitPrice = 11.95 " +
 "WHERE UnitPrice = 10.95 AND Format = 'Hardcover'";
int rowsUpdated = stmt.executeUpdate(query);
```

But what if you wanted to do the updates on a book-by-book basis? What if you only want to increase the price of your bestsellers, rather than every single book?

You would have to get the values from the database using a SELECT, then store the values in an array indexed somehow (perhaps with the primary key), then construct the appropriate UPDATE command strings, and call `executeUpdate()` one row at a time. Another option is to update the `ResultSet` directly.

When you create a `Statement` with concurrency set to `CONCUR_UPDATABLE`, you can modify the data in a result set and then apply your changes back to the database without having to issue another query.

In addition to the `getxxxx` methods we looked at for `ResultSet`—methods that get column values as integers, `Date` objects, `Strings`, etc.—there is an equivalent `updatexxxx` method for each type. And, just like the `getxxxx` methods, the `updatexxxx` methods can take either a `String` column name or an integer column index.

Let's rewrite the previous update example using an updatable `ResultSet`:

```

// We have a connection and we are in a try-catch block...
Statement stmt = // Scrollable
 conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, // and
 ResultSet.CONCUR_UPDATABLE); // updatable
String query = "SELECT UnitPrice from Book " +
 "WHERE Format = 'Hardcover'";
ResultSet rs = stmt.executeQuery(query); // Populate the ResultSet
while (rs.next()) {
 if (rs.getFloat("UnitPrice") == 10.95f) { // Check each row: if
 // unitPrice = 10.95
 rs.updateFloat("UnitPrice", 11.95f); // set it to 11.95
 rs.updateRow(); // and update the row
 // in the database
 }
}

```

Notice that after modifying the value of `UnitPrice` using the `updateFloat()` method, we called the method `updateRow()`. This method writes the current row to the database. This two-step approach ensures that all of the changes are made to the row before the row is written to the database. And you can change your mind with a `cancelRowUpdates()` method call.

[Table 12-11](#) summarizes methods that are commonly used with updatable `ResultSets` (whose concurrency type is set to `CONCUR_UPDATABLE`).

Let's look at the common methods used for altering database contents through the `ResultSet` in detail.

**TABLE 12-11** Methods Used with Updatable `ResultSets`

| Method                               | Purpose                                                                                                                                                                                                                                                                                            |
|--------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>void updateRow()</code>        | Updates the database with the contents of the current row of this <code>ResultSet</code> .                                                                                                                                                                                                         |
| <code>void deleteRow()</code>        | Deletes the current row from the <code>ResultSet</code> and the underlying database.                                                                                                                                                                                                               |
| <code>void cancelRowUpdates()</code> | Cancels any updates made to the current row of this <code>ResultSet</code> object. This method will effectively undo any changes made to the <code>ResultSet</code> row. If the <code>updateRow()</code> method was called before <code>cancelRowUpdates</code> , this method will have no effect. |
| <code>void moveToInsertRow()</code>  | Moves the cursor to a special row in the <code>ResultSet</code> set aside for performing an insert. You need to move to the insert row before updating the columns of the row with update methods and calling <code>insertRow()</code> .                                                           |
| <code>void insertRow()</code>        | Inserts the contents of the insert row into the database. Note that this method does not change the current <code>ResultSet</code> , so the <code>ResultSet</code> should be read again if you want the <code>ResultSet</code> to be consistent with the contents of the database.                 |
| <code>void moveToCurrentRow()</code> | Moves the cursor back to the current row from the insert row. If the cursor was not on the insert row, this method has no effect.                                                                                                                                                                  |

**public void updateRow() throws SQLException** This method updates the database with the contents of the current row of the `ResultSet`. There are a

couple of caveats for this method. First, the `ResultSet` must be from a SQL SELECT statement on a single table—a SQL statement that includes a JOIN or a SQL statement with two tables cannot be updated. Second, the `updateRow()` method should be called *before* moving to the next row. Otherwise, the updates to the current row may be lost.

So the typical use for this method is to update the contents of a row using the appropriate `updateXXXX()` methods and then update the database with the contents of the row using the `updateRow()` method. For example, in this fragment, we are updating the `UnitPrice` of a row to \$11.95:

```
rs.updateFloat("UnitPrice", 11.95f); // Set the price to 11.95
rs.updateRow(); // Update the row in the DB
```

**public boolean rowUpdated() throws SQLException** This method returns true if the current row was updated. Note that not all databases can detect updates. However, JDBC provides a method in `DatabaseMetaData` to determine if updates are detectable, `DatabaseMetaData.updatesAreDetected(int type)`, where the type is one of the `ResultSet` types—`TYPE_SCROLL_INSENSITIVE`, for example. We will cover the `DatabaseMetaData` interface and its methods a little later in this section.

```
if (rs.rowUpdated()) { // Has this row been modified?
 out.println("Row: " + rs.getRow() + " updated.");
}
```

**public void cancelRowUpdates() throws SQLException** This method allows you to “back out” changes made to the row. This method is important, because the `updateXXXX` methods should not be called twice on the same column. In other words, if you set the value of `UnitPrice` to 11.95 in the previous example and then decided to switch the price back to 10.95, calling the `updateFloat()` method again can lead to unpredictable results. So the better approach is to call `cancelRowUpdates()` before changing the value of a column a second time.

```
boolean priceRollback = ...; // Price rollback set somewhere else
while (rs.next()) {
 if (rs.getFloat("UnitPrice") == 10.95f) {
 rs.updateFloat("UnitPrice", 11.95f);
 }
 if (priceRollback) { // If priceRollback is true
 rs.cancelRowUpdates(); // Rollback changes to this row
 } else {
 rs.updateRow(); // else, commit this row to the DB
 }
}
```

**public void deleteRow() throws SQLException** This method will remove the current row from the `ResultSet` and from the underlying database. The row in the database is removed (similar to the result of a `DELETE` statement).

```
rs.last();
rs.deleteRow(); // Delete the last row.
```

What happens to the `ResultSet` after a `deleteRow()` method depends on whether the `ResultSet` can detect deletions. And this ability depends on the JDBC driver.

The `DatabaseMetaData` interface can be used to determine if the `ResultSet` can detect deletions:

```

int type = ResultSet.TYPE_SCROLL_INSENSITIVE; // Scrollable ResultSet
DatabaseMetaData dbmd = conn.getMetaData(); // Get meta data about
 // the driver and DB
if (dbmd.deletesAreDetected(type)) { // Returns false if deleted rows
 // are removed from the ResultSet
 while (rs.next()) { // Iterate through the ResultSet
 if (rs.rowDeleted()) { // Deleted rows are flagged, but
 continue; // not removed, so skip them
 } else {
 // process the row
 }
 } else {
 // Close the ResultSet and re-run the query
 }
}

```

In general, to maintain an up-to-date `ResultSet` after a deletion, the `ResultSet` should be re-created with a query.

Deleting the current row does not move the cursor—it remains on the current row—so if you deleted row 1, the cursor is still positioned at row 1. However, if the deleted row was the last row, then the cursor is positioned after the last row. Note that there is no undo for `deleteRow()`, at least, not by default.

**public boolean rowDeleted() throws SQLException** As described earlier, when a `ResultSet` can detect deletes, the `rowDeleted()` method is used to indicate a row has been deleted but remains as a part of the `ResultSet` object. For example, suppose that we deleted the second row of the `Customer` table. Printing the results (after the delete) to the console would look like [Figure 12-7](#).

## FIGURE 12-7

A `ResultSet` after `deleteRow()` is called on the second row



```
String query = "SELECT * FROM Customer";
ResultSet rs = stmt.executeQuery(query);
rs.next();
rs.next();
rs.deleteRow();
```

ResultSet

|      |             |           |                         |              |
|------|-------------|-----------|-------------------------|--------------|
| 5000 | John        | Smith     | john.smith@verizon.net  | 555-340-1230 |
| null | null        | null      | null                    | null         |
| 5002 | Bob         | Collins   | bob.collins@yahoo.com   | 555-012-3456 |
| 5003 | Rebecca     | Mayer     | rebecca.mayer@gmail.com | 555-205-8212 |
| 5006 | Anthony     | Clark     | anthony.clark@gmail.com | 555-256-1901 |
| 5007 | Judy        | Sousa     | judy.sousa@verizon.net  | 555-751-1207 |
| 5008 | Christopher | Patriquin | patriquin@yahoo.com     | 555-316-1803 |
| 5009 | Deborah     | Smith     | deb.smith@comcast.net   | 555-256-3421 |
| 5010 | Jennifer    | McGinn    | jmcginn@comcast.net     | 555-250-0918 |

So if you are working with a `ResultSet` that is being passed around between methods and shared across classes, you might use `rowDeleted()` to detect if the current row contains valid data.

**Updating Columns Using Objects** An interesting aspect of the `getObject()` and `updateObject()` methods is that they retrieve a column as a Java object.

And because every Java object can be turned into a `String` using the object's `toString()` method, you can retrieve the value of any column in the database and print the value to the console as a `String`, as you saw in the section "Printing a Report."

Going the other way, toward the database, you can also use `Strings` to update almost every column in a `ResultSet`. All of the most common SQL types—`integer`, `float`, `double`, `long`, and `date`—are wrapped by their representative Java object: `Integer`, `Float`, `Double`, `Long`, and `java.sql.Date`. Each of these objects has a method `valueOf()` that takes a `String`.

The `updateObject()` method takes two arguments: the first, a column name (`String`) or column index; and the second, an `Object`. We can pass a `String` as the `Object` type, and as long as the `String` meets the requirements of the `valueOf()` method for the column type, the `String` will be properly converted and stored in the database as the desired SQL type.

For example, suppose that we are going to update the publish date (`PubDate`) of one of our books:

```
// We have a connection and we are in a try-catch block...
Statement stmt =
 conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
 ResultSet.CONCUR_UPDATABLE);
String query = "SELECT * FROM Book WHERE ISBN='142311339X'";
ResultSet rs = stmt.executeQuery(query);
rs.next();
rs.updateObject("PubDate", "2005-04-23"); // Update PubDate using
 // a String date
rs.updateRow(); // Update this row
```

The `String` we passed meets the requirements for `java.sql.Date`, "yyyy-[m]m-[d]d," so the `String` is properly converted and stored in the database as the SQL Date value: 2005-04-23. Note this technique is limited to those SQL types that can be converted to and from a `String`, and if the `String` passed to the `valueOf()` method for the SQL type of the column is not properly formatted for the Java object, an `IllegalArgumentException` is thrown.

## Inserting New Rows Using a ResultSet

In the last section, we looked at modifying the existing column data in a `ResultSet` and removing existing rows. In our final section on `ResultSets`, we'll look at how to create and insert a new row. First, you must have a valid `ResultSet` open, so typically, you have performed some query. `ResultSet` provides a special row, called the insert row, that you are actually modifying (updating) before performing the insert. Think of the insert row as a buffer where you can modify an empty row of your `ResultSet` with values.

Inserting a row is a three-step process, as shown in [Figure 12-8](#): First (1) move to the special insert row, then (2) update the values of the columns for the new row, and finally (3) perform the actual insert (write to the underlying database). The existing `ResultSet` is not changed—you must rerun your query to see the underlying changes in the database. However, you can insert as many rows as you like. Note that each of these methods throws a `SQLException` if the concurrency type of the result set is set to `CONCUR_READ_ONLY`. Let's look at the methods before we look at example code.

### FIGURE 12-8

The `ResultSet` insert row

```

String query = "SELECT AuthorID, FirstName, LastName FROM Author";
ResultSet rs = stmt.executeQuery(query);
----- rs.next();
| - rs.moveToInsertRow();
| rs.updateInt("AuthorID", 1055);
| rs.updateString("FirstName", "Tom");
| rs.updateString("LastName", "McGinn");
1 | rs.insertRow();
| rs.moveToCurrentRow();

```

①

③

ResultSet

|      |          |          |
|------|----------|----------|
| 1000 | Rick     | Riordan  |
| 1001 | Nancy    | Farmer   |
| 1002 | Ally     | Condie   |
| 1003 | Cressida | Cowell   |
| 1004 | Lauren   | St. John |
| 1005 | Erin     | Colfer   |

②

insert row

|      |     |        |
|------|-----|--------|
| 1055 | Tom | McGinn |
|------|-----|--------|

**public void moveToInsertRow() throws SQLException** This method moves the cursor to insert a row buffer. Wherever the cursor was when this method was

called is remembered. After calling this method, the appropriate updater methods are called to update the values of the columns.

```
rs.moveToInsertRow();
```

**public void insertRow() throws SQLException** This method writes the insert row buffer to the database. Note that the cursor must be on the insert row when this method is called. Also, note that each column must be set to a value before the row is inserted in the database or a SQLException will be thrown. The insertRow( ) method can be called more than once—however, the insertRow follows the same rules as a SQL INSERT command. Unless the primary key is auto-generated, two inserts of the same data will result in a SQLException (duplicate primary key).

```
rs.insertRow();
```

**public void moveToCurrentRow() throws SQLException** This method returns the result set cursor to the row the cursor was on before the moveToInsertRow( ) method was called.

Let's look at a simple example, where we will add a new row in the Author table:

```

// We have a connection and we are in a try-catch block...
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
 ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT AuthorID, FirstName, LastName
 FROM Author");
rs.next();
rs.moveToInsertRow(); // Move the special insert row
rs.updateInt("AuthorID", 1055); // Create an author ID
rs.updateString("FirstName", "Tom"); // Set the first name
rs.updateString("LastName", "McGinn"); // Set the last name
rs.insertRow(); // Insert the row into the database
rs.moveToCurrentRow(); // Move back to the current row in
 // ResultSet

```

## Getting Information about a Database Using DatabaseMetaData

Note: On the real exam, you might not get any questions about DatabaseMetaData.

In the example we are using in this chapter, Bob's Books, we know quite a lot about the tables, columns, and relationships between the tables because we had that nifty data model earlier. But what if that were not the case? This section covers DatabaseMetaData, an interface that provides a significant amount of information about the database itself. This topic is fairly advanced stuff and is not on the exam, but it is provided here to give you an idea about how you can use metadata to build a model of a database without having to know anything about the database in advance.

Recall that the Connection object we obtained from DriverManager is an object that represents an actual connection with the database. And while the Connection object is primarily used to create Statement objects, there are a couple of important methods to study in the Connection interface. A Connection can be used to obtain information *about* the database as well. This data is called “metadata,” or “data about data.”

One of Connection’s methods returns a DatabaseMetaData object instance,

through which we can get information about the database, about the driver, and about transaction semantics that the database and JDBC driver support.

To obtain an instance of a `DatabaseMetaData` object, we use `Connection`'s `getMetaData()` method:

```
String url = "jdbc:derby://localhost:1521/BookSellerDB";
String user = "bookguy";
String pwd = "$3lleR";
try {
 Connection conn = DriverManager.getConnection(url, user, pwd);
 DatabaseMetaData dbmd = conn.getMetaData(); // Get the database
 // metadata
} catch (SQLException se) { }
```

`DatabaseMetaData` is a comprehensive interface, and through an object instance, we can determine a great deal about the database and the supporting driver. Most of the time, as a developer, you aren't coding against a database blindly and know the capabilities of the database and the driver before you write any code. Still, it is helpful to know that you can use `getObject` to return the value of the column, regardless of its type—very useful when all you want to do is create a report. We'll look at an example.

Here are a few methods we will highlight:

- **`getColumns()`** Returns a description of columns in a specified catalog and schema
- **`getProcedures()`** Returns a description of the stored procedures in a given catalog and schema
- **`getDriverName()`** Returns the name of the JDBC driver
- **`getDriverVersion()`** Returns the version number of the JDBC driver as a string
- **`supportsANSI92EntryLevelSQL()`** Returns a boolean true if this database supports ANSI92 entry-level grammar

It is interesting to note that `DatabaseMetaData` methods also use `ResultSet`

objects to return data about the database. Let's look at these methods in more detail.

**public ResultSet getColumns(String catalog, String schemaPattern, String tableNamePattern, String columnNamePattern) throws SQLException**

**SQLException** This method is one of the best all-purpose data retrieval methods for details about the tables and columns in your database. Before we look at a code sample, it might be helpful to define catalogs and schemas. In a database, a schema is an object that enforces the integrity of the tables in the database. The schema name is generally the name of the person who created the database. In our examples, the BookGuy database holds the collection of tables and is the name of the schema. Databases may have multiple schemas stored in a catalog.

In this example, using the Java DB database as our sample database, the catalog is null and our schema is “BOOKGUY”, and we are using a SQL catch-all pattern “%” for the table and column name patterns, like the “\*” character you are probably used to with file systems like Windows. Thus, we are going to retrieve all of the tables and columns in the schema. Specifically, we are going to print out the table name, column name, the SQL data type for the column, and the size of the column. Note that here we used uppercase column identifiers. These are the column names verbatim from the JavaDoc, but in truth, they are not case-sensitive either, so “Table\_Name” would have worked just as well. Also, the JavaDoc specifies the column index for these column headings, so we could have also used `rs.getString(3)` to get the table name.

```

String url = "jdbc:derby://localhost:1521/BookSellerDB";
String user = "bookguy";
String pwd = "$3lleR";
try {
 Connection conn = DriverManager.getConnection(url, user, pwd);
 DatabaseMetaData dbmd = conn.getMetaData();
 ResultSet rs
 = dbmd.getColumns(null, "BOOKGUY", "%", "%"); // Get a ResultSet
 // for any catalog (null)
 // in the BOOKGUY schema
 // for all tables (%)
 // for all columns (%)

 while (rs.next()) {
 out.print("Table Name: " + rs.getString("TABLE_NAME") + " ");
 out.print("Column_Name: " + rs.getString("COLUMN_NAME") + " ");
 out.print("Type_Name: " + rs.getString("TYPE_NAME") + " ");
 out.println("Column Size " + rs.getString("COLUMN_SIZE"));
 }
} catch (SQLException se) {
 out.println("SQLException: " + se);
}

```

Running this code produces output something like this:

Table Name: AUTHOR Column\_Name: AUTHORID Type\_Name: INTEGER Column Size 10 Primary Key

Table Name: AUTHOR Column\_Name: FIRSTNAME Type\_Name: VARCHAR Column Size 20

Table Name: AUTHOR Column\_Name: LASTNAME Type\_Name: VARCHAR Column Size 20

Table Name: BOOK Column\_Name: ISBN Type\_Name: VARCHAR Column Size 10 Primary Key

Table Name: BOOK Column\_Name: TITLE Type\_Name: VARCHAR Column Size 100

Table Name: BOOK Column\_Name: PUBDATE Type\_Name: DATE Column Size 10

Table Name: BOOK Column\_Name: FORMAT Type\_Name: VARCHAR Column Size 30

Table Name: BOOK Column\_Name: UNITPRICE Type\_Name: DOUBLE Column Size 52

Table Name: BOOKS\_BY\_AUTHOR Column\_Name: AUTHORID Type\_Name: INTEGER Column Size 10

Table Name: BOOKS\_BY\_AUTHOR Column\_Name: ISBN Type\_Name: VARCHAR Column Size 10

Table Name: CUSTOMER Column\_Name: CUSTOMERID Type\_Name: INTEGER Column Size 10 Primary Key

Table Name: CUSTOMER Column\_Name: FIRSTNAME Type\_Name: VARCHAR Column Size 30

Table Name: CUSTOMER Column\_Name: LASTNAME Type\_Name: VARCHAR Column Size 30

Table Name: CUSTOMER Column\_Name: EMAIL Type\_Name: VARCHAR Column Size 40

Table Name: CUSTOMER Column\_Name: PHONE Type\_Name: VARCHAR Column Size 15

**public ResultSet getProcedures(String catalog, String schemaPattern, String procedureNamePattern) throws SQLException** Stored procedures are

functions that are sometimes built into a database and often defined by a database developer or database admin. These functions can range from data cleanup to complex queries. This method returns a result set that contains descriptive information about the stored procedures for a catalog and schema. In the example code, we will use null for the catalog name and schema pattern. The null indicates that we do not wish to narrow the search (effectively, the same as using a catch-all “%” search). Note that this example is returning the name of every stored procedure in the database. A little later, we’ll look at how to actually call a stored procedure.

```
try {
 Connection conn = ...
 DatabaseMetaData dbmd = conn.getMetaData();
 ResultSet rs =
 dbmd.getProcedures(null, null, "%"); // Get a ResultSet of all
 // the stored procedures
 // in any catalog (null)
 // in any schema (null)
 // with wildcard name (%)

 while(rs.next()) {
 out.println("Procedure Name: " + rs.getString("PROCEDURE_NAME"));
 }
} catch (SQLException se) { }
```

Note that the output from this code fragment is highly database dependent. Here is sample output from the Derby (JavaDB) database that ships with the JDK:

Procedure Name: INSTALL\_JAR  
Procedure Name: REMOVE\_JAR  
Procedure Name: REPLACE\_JAR  
Procedure Name: SYSCS\_BACKUP\_DATABASE  
Procedure Name: SYSCS\_BACKUP\_DATABASE\_AND\_ENABLE\_LOG\_ARCHIVE\_MODE  
Procedure Name: SYSCS\_BACKUP\_DATABASE\_AND\_ENABLE\_LOG\_ARCHIVE\_MODE\_NOWAIT  
Procedure Name: SYSCS\_BACKUP\_DATABASE\_NOWAIT  
Procedure Name: SYSCS\_BULK\_INSERT

**public String getDriverName() throws SQLException** This method simply returns the name of the JDBC driver as a string. This method would be useful to log at the start of the application, as you'll see in the next section.

```
System.out.println("getDriverName: " + dbmd.getDriverName());
```

Obviously, the name of the driver depends on the JDBC driver you are using. Again, with the Derby database and JDBC driver, the output from this method looks something like this:

```
getDriverName: Apache Derby Network Client JDBC Driver
```

**public String getDriverVersion() throws SQLException** This method returns the JDBC driver version number as a string. This information and the driver name would be good to log in at startup of an application.

```
Logger logger = Logger.getLogger("com.cert.DatabaseMetaDataTest");
Connection conn = ...
DatabaseMetaData dbmd = conn.getMetaData();
logger.log(Level.INFO, "Driver Version: {0}", dbmd.getDriverVersion());
logger.log(Level.INFO, "Driver Name: {0}", dbmd.getDriverName());
```

Statements written to the log are generally recorded in a log file, but depending on the IDE, they can also be written to the console. In NetBeans, for example, the log statements look something like this in the console:

```
Sep 23, 2012 3:55:39 PM com.cert.DatabaseMetaDataTest main
INFO: Driver Version: 10.8.2.2 - (1181258)
Sep 23, 2012 3:55:39 PM com.cert.DatabaseMetaDataTest main
INFO: Driver Name: Apache Derby Network Client JDBC Driver
```

#### **public boolean supportsANSI92EntryLevelSQL() throws**

**SQLException** This method returns true if the database and JDBC driver support ANSI SQL-92 entry-level grammar. Support for this level (at a minimum) is a requirement for JDBC drivers (and, therefore, the database).

```
Connection conn = ...
DatabaseMetaData dbmd = conn.getMetaData();
if (!dbmd.supportsANSI92EntryLevelSQL()) {
 logger.log(Level.WARNING, "JDBC Driver does not meet minimum
 requirements for SQL-92 support");
}
```

## **When Things Go Wrong—Exceptions and Warnings**

Whenever you are working with a database using JDBC, there is a possibility that something can go wrong. A JDBC connection is typically through a socket to a database resource on the network. So already we have at least two possible points of failure—the network can be down and/or the database can be down. And that assumes everything else you are doing with your database is correct, that all your queries are perfect! Like other Java exceptions, **SQLException** is a way for your application to determine what the problem is and take action if necessary.

Let's look at the type of data you get from a **SQLException** through its methods.

**public String getMessage()** This method is actually inherited from `java.lang.Exception`, which **SQLException** extends from. This method returns the detailed reason why the exception was thrown. Often, the message contents `SQLState` and error code provide specific information about what went wrong.

**public String getSQLState()** The String returned by `getSQLState` provides a specific code and related message. SQLState messages are defined by the X/Open and SQL:2003 standards; however, it is up to the implementation to use these values. You can determine which standard your JDBC driver uses (or if it does not) through the `DatabaseMetaData.getSQLStateType()` method. Your implementation may also define additional codes specific to the implementation, so in either case, it is a good idea to consult your JDBC driver and database documentation. Because the SQLState messages and codes tend to be specific to the driver and database, the typical use of these in an application is limited to either logging messages or debugging information.

**public int getErrorCode()** Error codes are not defined by a standard and are thus implementation specific. They can be used to pass an actual error code or severity level, depending on the implementation.

**public SQLException getNextException()** One of the interesting aspects of `SQLException` is that the exception thrown could be the result of more than one issue. Fortunately, JDBC simply tacks each exception onto the next in a process called chaining. Typically, the most severe exception is thrown last, so it is the first exception in the chain.

You can get a list of all of the exceptions in the chain using the `getNextException()` method to iterate through the list. When the end of the list is reached, `getNextException()` returns a null. In this example, the `SQLExceptions`, `SQLState`, and vendor error codes are logged:

```
Logger logger = Logger.getLogger("com.example.MyClass");
try {
 // some JDBC code in a try block
 // ...
} catch (SQLException se) {
 while (se != null) {
 logger.log(Level.SEVERE, "----- SQLException -----");
 logger.log(Level.SEVERE, "SQLState: " + se.getSQLState());
 logger.log(Level.SEVERE, "Vendor Error code: " +
 se.getErrorCode());
 logger.log(Level.SEVERE, "Message: " + se.getMessage());
 se = se.getNextException();
 }
}
```

## Warnings

Although `SQLWarning` is a subclass of `SQLException`, warnings are silently chained to the JDBC object that reported them. This is probably one of the few times in Java where an object that is part of an exception hierarchy is not thrown as an exception. The reason is that a warning is not an exception per se.

Warnings can be reported on `Connection`, `Statement`, and `ResultSet` objects.

For example, suppose that we mistakenly set the result-set type to `TYPE_SCROLL_SENSITIVE` when creating a `Statement` object. This does not create an exception; instead, the database will handle the situation by chaining a `SQLWarning` to the `Connection` object and resetting the type to `TYPE_FORWARD_ONLY` (the default) and continue on. Everything would be fine, of course, until we tried to position the cursor, at which point a `SQLException` would be thrown. And, like `SQLException`, you can retrieve warnings from the `SQLWarning` object using the `getNextWarning()` method.

```

Connection conn =
 DriverManager.getConnection("jdbc:derby://localhost:1527/BookSellerDB",
 "bookguy", "S3ll3R");
Statement stmt =
 conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
 ResultSet.CONCUR_UPDATABLE);
String query = "SELECT * from Book WHERE Book.Format = 'Hardcover'";
ResultSet rs = stmt.executeQuery(query);
SQLWarning warn = conn.getWarnings(); // Get any SQLWarnings
while (warn != null) { // If there is a SQLWarning, print it
 out.println("SQLState: " + warn.getSQLState());
 out.println("Message: " + warn.getMessage());
 warn = warn.getNextWarning(); // Get the next warning
}

```

Connection objects will add warnings (if necessary) until the Connection is closed or until the `clearWarnings()` method is called on the Connection instance. The `clearWarnings()` method sets the list of warnings to null until another warning is reported for this Connection object.

Statements and ResultSets also generate SQLWarnings, and these objects have their own `clearWarnings()` methods. Statement warnings are cleared automatically when a statement is reexecuted, and ResultSet warnings are cleared each time a new row is read from the result set.

The following sections summarize the methods associated with `SQLWarnings`.

**SQLWarning getWarnings() throws SQLException** This method gets the first `SQLWarning` object or returns null if there are no warnings for this Connection, Statement, or ResultSet object. A `SQLException` is thrown if the method is called on a closed object.

**void clearWarnings() throws SQLException** This method clears and resets the current set of warnings for this Connection, Statement, or ResultSet object.

A `SQLException` is thrown if the method is called on a closed object.

## Properly Closing SQL Resources

In this chapter, we have looked at some very simple examples where we create a `Connection` and `Statement` and a `ResultSet` all within a single `try` block and catch any `SQLExceptions` thrown. What we have not done so far is properly close these resources. The reality is that it is probably less important for such small examples, but for any code that uses a resource, like a socket, or a file, or a JDBC database connection, closing the open resources is a good practice.

It is also important to know when a resource is closed automatically. Each of the three major JDBC objects—`Connection`, `Statement`, and `ResultSet`—has a `close()` method to explicitly close the resource associated with the object and explicitly release the resource. We hope by now you also realize that the objects have a relationship with each other, so if one object executes `close()`, it will have an impact on the other objects. The following table should help explain this.

| Method Call                            | Has the Following Action(s)                                                                                                                                       |
|----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Connection.close()</code>        | Releases the connection to the database.<br>Closes any <code>Statement</code> created from this <code>Connection</code> .                                         |
| <code>Statement.close()</code>         | Releases this <code>Statement</code> resource.<br>Closes any open <code>ResultSet</code> associated with this <code>Statement</code> .                            |
| <code>ResultSet.close()</code>         | Releases this <code>ResultSet</code> resource. Note that any <code>ResultSetMetaData</code> objects created from the <code>ResultSet</code> are still accessible. |
| <code>Statement.executeUpdate()</code> | Any <code>ResultSet</code> associated with a previous <code>Statement</code> execution is automatically closed.                                                   |

It is also a good practice to minimize the number of times you close and re-create `Connection` objects. As a rule, creating the connection to the database and

passing the username and password credentials for authentication is a relatively expensive process, so performing the activity once for every SQL query can cause code to execute slowly. In fact, typically, database connections are created in a pool and connection instances are handed out to applications as needed, rather than allowing or requiring individual applications to create them.

Statement objects are less expensive to create. There are ways to precompile SQL statements using a PreparedStatement, which reduces the overhead associated with creating SQL query strings and sending those strings to the database for execution, but understanding PreparedStatement is no longer on the exam.

ResultSets are the least expensive of the objects to create, and as you saw in the section “ResultSets,” for results from a single table, you can use the ResultSet to update, insert, and delete rows, so it can be very efficient to use a ResultSet.

Let’s look at one of our previous examples, where we used a Connection, a Statement, and a ResultSet, and rewrite this code to close the resources properly.

```
Connection conn = null;
String url, user, pwd; // These are populated somewhere else
try {
 conn = DriverManager.getConnection(url, user, pwd);
 Statement stmt = conn.createStatement();
 ResultSet rs = stmt.executeQuery("SELECT * FROM Customer");
 // ... process the results
 // ...
 if (rs != null && stmt != null) {
 rs.close(); // Attempt to close the ResultSet
 stmt.close(); // Attempt to close the Statement
 }
} catch (SQLException se) {
 out.println("SQLException: " + se);
} finally {
 try {
 if (conn != null) {
 conn.close(); // Close the Connection
 }
 } catch (SQLException sec) {
 out.println("Exception closing connection!");
 }
}
```

Notice all the work we have to go through to close the connection—we first need to make sure we actually got an object and not a null, and then we need to try the `close()` method inside of another `try` inside of the `finally` block! Fortunately, there is an easier way....

## Using try-with-resources to Close Connections, Statements, and ResultSets

One of the most useful changes in Java SE 7 (JDK 7) was a number of small modifications to the language, including a new `try` statement to support automatic resource management. This language change is called `try-with-resources`, and its longer name belies how much simpler it makes writing code with resources that should be closed. The `try-with-resources` statement will automatically call the `close()` method on any resource declared in the parentheses at the end of the `try` block.

There is a caveat: A resource declared in the `try-with-resource` statement must implement the `AutoCloseable` interface. One of the changes for JDBC in Java SE 7 (JDBC 4.1) was the modification of the API so that `Connection`, `Statement`, and `ResultSet` all extend the `AutoCloseable` interface and support automatic resource management. So we can rewrite our previous code example using `try-with-resources`:

```
String url, user, pwd; // These are populated somewhere else
try (Connection conn = DriverManager.getConnection(url, user, pwd)) {
 Statement stmt = conn.createStatement();
 ResultSet rs = stmt.executeQuery("SELECT * FROM Customer");
 // ... process the results
 // ...
 if (rs != null && stmt != null) {
 rs.close(); // Attempt to close the ResultSet
 stmt.close(); // Attempt to close the Statement
 }
} catch (SQLException se) {
 out.println("SQLException: " + se);
}
```

Notice that we must include the object type in the declaration inside of the

parentheses. The following will throw a compilation error:

```
try (conn = DriverManager.getConnection(url, user, pwd)) {
```

The `try-with-resources` can also be used with multiple resources, so you could include the `Statement` declaration in the `try` as well:

```
try (Connection conn = DriverManager.getConnection(url, user, pwd);
 Statement stmt = conn.createStatement()) {
```

Note that when more than one resource is declared in the `try-with-resources` statement, the resources are closed in the reverse order of their declaration—so `stmt.close()` will be called first, followed by `conn.close()`.

It probably makes sense that, if an exception is thrown from the `try` block, the exception will be caught by the `catch` statement, but what happens to exceptions thrown as a result of closing the resources in the `try-with-resources` statement? Any exceptions thrown as a result of closing resources at the end of the `try` block are suppressed, if there was also an exception thrown in the `try` block. These exceptions can be retrieved from the exception thrown by calling the `getSuppressed()` method on the exception thrown.

For example:

```
} catch (SQLException se) {
 out.println("SQLException: " + se);
 Throwable[] suppressed = se.getSuppressed(); // Get an array of
 // suppressed
 // exceptions
 for (Throwable t: suppressed) { // Iterate through the array
 out.println("Suppressed exception: " + t);
 }
}
```

## CERTIFICATION SUMMARY

---

## Core JDBC API

Remember that the JDBC API is a set of interfaces with one important concrete class, the `DriverManager` class. You write code using the well-defined set of JDBC interfaces, and the provider of your JDBC driver writes code implementations of those interfaces. The key (and, therefore, required) interfaces a JDBC driver must implement include `Driver`, `Connection`, `Statement`, and `ResultSet`.

The driver provider will also implement an instance of `DatabaseMetaData`, which you use to invoke a method to query the driver for information about the database and JDBC driver. One important piece of information is if the database is SQL-92 compliant, and there are a number of methods that begin with “supports” to determine the capabilities of the driver. One important method is `supportsResultSetType()`, which is used to determine if the driver supports scrolling result sets.

## DriverManager

The `DriverManager` is one of the few concrete classes in the JDBC API, and you will recall that the `DriverManager` is a factory class—using the `DriverManager`, you construct instances of `Connection` objects. In reality, the `DriverManager` simply holds references to registered JDBC drivers, and when you invoke the `getConnection()` method with a JDBC URL, the `DriverManager` passes the URL to each driver in turn. If the URL matches a valid driver, host, port number, username, and password, then that driver returns an instance of a `Connection` object. Remember that the JDBC URL is simply a string that encodes the information required to make a connection to a database.

How a JDBC driver is registered with the `DriverManager` is also important. In JDBC 4.0 and later, the driver jar file simply needs to be on the classpath, and the `DriverManager` will take care of finding the driver’s `Driver` class implementation and load that. JDBC 3.0 and earlier, require that the driver’s `Driver` class implementation be manually loaded using the `Class.forName()` method with the fully qualified class name of the class.

## Statements and ResultSets

The most important use of a database is clearly using SQL statements and queries to create, read, update, and delete database records. The `Statement` interface provides the methods needed to create SQL statements and execute

them. Remember that there are three different Statement methods to execute SQL queries: one that returns a result set, `executeQuery()`; one that returns an affected row count, `executeUpdate()`; and one general-purpose method that returns a boolean to indicate if the query produced a result set, `execute()`.

`ResultSet` is the interface used to read columns of data returned from a query, one row at a time. `ResultSet` objects represent a snapshot (a copy) of the data returned from a query, and there is a cursor that points to just above the first row when the results are returned. Unless you created a `Statement` object using the `Connection.createStatement(int, int)` method that takes `resultSetType` and `resultSetConcurrency` parameters, `ResultSets` are not updatable and only allow the cursor to move forward through the results. However, if your database supports it, you can create a `Statement` object with a type of `ResultSet.TYPE_SCROLL_INSENSITIVE` and/or a concurrency of `ResultSet.CONCUR_UPDATABLE`, which allows any result set created with the `Statement` object to position the cursor anywhere in the results (scrollable) and allows you to change the value of any column in any row in the result set (updatable). Finally, when using a `ResultSet` that is scrollable, you can determine the number of rows returned from a query—and this is the only way to determine the row count because there is no “`rowCount`” method.

`SQLException` is the base class for exceptions thrown by JDBC, and because one query can result in a number of exceptions, the exceptions are chained. To determine all of the reasons a method call returned a `SQLException`, you must iterate through the exception by calling the `getNextException()` method. JDBC also keeps track of warnings for methods on `Connection`, `Statement`, and `ResultSet` objects using a `SQLWarning` exception type. Like `SQLException`, `SQLWarning` is silently chained to the object that caused the warning—for example, suppose that you attempt to create a `Statement` object that supports the scrollable `ResultSet`, but the database does not support that type. A `SQLWarning` will be added to the `Connection` object (the `Connection.createStatement(int, int)` method creates a `Statement` object). The `getWarnings()` method is used to return any `SQLWarnings`.

One of the important additions to Java SE 7 was the `try-with-resources` statement, and all of the JDBC interfaces have been updated to support the new `AutoCloseable` interface. However, bear in mind that there is an order of precedence when closing `Connections`, `Statements`, and `ResultSets`. So when a `Connection` is closed, any `Statement` created from that `Connection` is also closed, and likewise, when a `Statement` is closed, any `ResultSet` created using that `Statement` is also closed. And attempting to invoke a method on a closed

object will result in a `SQLException`!



## TWO-MINUTE DRILL

Here are some of the key points from the certification objectives in this chapter.

### Core Interfaces of the JDBC API (OCP Objective 11.1)

- To be compliant with JDBC, driver vendors must provide implementations for the key JDBC interfaces: `Driver`, `Connection`, `Statement`, and `ResultSet`.
- `DatabaseMetaData` can be used to determine which SQL-92 level your driver and database support.
- `DatabaseMetaData` provides methods to interrogate the driver for capabilities and features.

### Connect to a Database Using DriverManager (OCP Objective 11.2)

- The JDBC API follows a factory pattern, where the `DriverManager` class is used to construct instances of `Connection` objects.
- The JDBC URL is passed to each registered driver, in turn, in an attempt to create a valid `Connection`.
- Identify the Java statements required to connect to a database using JDBC.
- JDBC 3.0 (and earlier) drivers must be loaded prior to their use.
- JDBC 4.0 drivers just need to be part of the classpath, and they are automatically loaded by the `DriverManager`.

### Submit Queries and Read Results from the Database (OCP Objective 11.3)

- The `next()` method must be called on a `ResultSet` before reading the

first row of results.

- When a Statement execute() method is executed, any open ResultSets tied to that Statement are automatically closed.
- When a statement is closed, any related ResultSets are also closed.
- ResultSet column indexes are numbered from 1, not 0.
- The default ResultSet is not updatable (read-only), and the cursor moves forward only.
- A ResultSet that is scrollable and updatable can be modified, and the cursor can be positioned anywhere within the ResultSet.
- ResultSetMetaData can be used to dynamically discover the number of columns and their type returned in a ResultSet.
- ResultSetMetaData does not have a row count method. To determine the number of rows returned, the ResultSet must be scrollable.
- ResultSet fetch size can be controlled for large data sets; however, it is a hint to the driver and may be ignored.
- SQLExceptions are chained. You must iterate through the exceptions thrown to get all of the reasons why an exception was thrown.
- SQLException also contains database-specific error codes and status codes.
- The executeQuery method is used to return a ResultSet (SELECT).
- The executeUpdate method is used to update data, to modify the database, and to return the number of rows affected (INSERT, UPDATE, DELETE, and DDLs).
- The execute method is used to perform any SQL command. A boolean true is returned when the query produces a ResultSet and false when there are no results, or if the result is an update count.
- There is an order of precedence in the closing of Connections, Statements, and ResultSets.
- Using the try-with-resources statement, you can close Connections, Statements, and ResultSets automatically (they implement the new AutoCloseable interface in Java SE 7).
- When a Connection is closed, all of the related Statements and ResultSets are closed.

# Q SELF TEST

1 Given:

```
String url = "jdbc:mysql://SolDBServer/soldb";
String user = "sysEntry";
String pwd = "fo0B3@r";
// INSERT CODE HERE
Connection conn = DriverManager.getConnection(url, user, pwd);
```

Assuming "org.gjt.mm.mysql.Driver" is a legitimate class, which line, when inserted at // INSERT CODE HERE, will correctly load this JDBC 3.0 driver?

- A. DriverManager.registerDriver("org.gjt.mm.mysql.Driver");
  - B. Class.forName("org.gjt.mm.mysql.Driver");
  - C. DatabaseMetaData.loadDriver("org.gjt.mm.mysql.Driver");
  - D. Driver.connect("org.gjt.mm.mysql.Driver");
  - E. DriverManager.getDriver("org.gjt.mm.mysql.Driver");
- 2 Given that you are working with a JDBC 4.0 driver, which three are required for this JDBC driver to be compliant?
- A. Must include a META-INF/services/java.sql.Driver file
  - B. Must provide implementations of Driver, Connection, Statement, and ResultSet interfaces
  - C. Must support scrollable ResultSets
  - D. Must support updatable ResultSets
  - E. Must support transactions
  - F. Must support the SQL99 standard

G. Must support `PreparedStatement` and `CallableStatement`

3 Which three are available through an instance of `DatabaseMetaData`?

- A. The number of columns returned
- B. The number of rows returned
- C. The name of the JDBC driver
- D. The default transaction isolation level
- E. The last query used
- F. The names of stored procedures in the database

4 Given:

```
try {
 Statement stmt = conn.createStatement();
 String query =
 "SELECT * FROM Author WHERE LastName LIKE 'Rand%';
 ResultSet rs = stmt.executeQuery(query); // Line X
 if (rs == null) { // Line Y
 System.out.println("No results");
 } else {
 System.out.println(rs.getString("FirstName"));
 }
} catch (SQLException se) {
 System.out.println("SQLException");
}
```

Assuming a `Connection` object has already been created (`conn`) and that the query produces a valid result, what is the result?

- A. Compiler error at line X
- B. Compiler error at line Y
- C. No result
- D. The first name from the first row that matches ‘Rand%’

- E. SQLException
- F. A runtime exception

5 Given the SQL query:

```
String query = "UPDATE Customer SET EMail='John.Smith@comcast.net'
 WHERE CustomerID = 5000";
```

Assuming this is a valid SQL query and there is a valid connection object (conn), which will compile correctly and execute this query?

- A. Statement stmt = conn.createStatement();  
 stmt.executeQuery(query);
- B. Statement stmt = conn.createStatement(query);  
 stmt.executeUpdate();
- C. Statement stmt = conn.createStatement();  
 stmt.setQuery(query);  
 stmt.execute();
- D. Statement stmt = conn.createStatement();  
 stmt.execute(query);
- E. Statement stmt = conn.createStatement();  
 ResultSet rs = stmt.executeUpdate(query);

6 Given:

```

try {
 ResultSet rs = null;
 try (Statement stmt = conn.createStatement()) { // line X
 String query = "SELECT * from Customer";
 rs = stmt.executeQuery(query); // line Y
 } catch (SQLException se) {
 System.out.println("Illegal query");
 }
 while (rs.next()) {
 // print customer names
 }
} catch (SQLException se) {
 System.out.println("SQLException");
}

```

And assuming a valid `Connection` object (`conn`) and that the query will return results, what is the result?

- A. The customer names will be printed out
  - B. Compiler error at line X
  - C. Illegal query
  - D. Compiler error at line Y
  - E. `SQLException`
  - F. Runtime exception
- 7 Which interfaces must a vendor implement to be JDBC compliant? (Choose all that apply.)
- A. `Query`
  - B. `Driver`
  - C. `ResultSet`
  - D. `Statement`

E. Connection

F. SQLException

- 8 Given this code fragment:

```
Statement stmt = conn.createStatement();
ResultSet rs;
String query = "<QUERY HERE>";
stmt.execute(query);
if ((rs = stmt.getResultSet()) != null) {
 System.out.println("Results");
}
if (stmt.getUpdateCount() > -1) {
 System.out.println("Update");
}
```

Assuming each query is valid and that all tables have valid row data, which query statements entered into <QUERY HERE> produce the output that follows the query string (in the following answer? (Choose all that apply.)

A. "SELECT \* FROM Customer"

Results

B. "INSERT INTO Book VALUES ('1023456789', 'One Night in Paris',  
'1984-10-20',  
'Hardcover', 13.95)"

Update

C. "UPDATE Customer SET Phone = '555-234-1021' WHERE CustomerID = 101"

Update

D. "SELECT Author.LastName FROM Author"

Results

E. "DELETE FROM Book WHERE ISBN = '1023456789'"

Update

9 Which are true about queries that throw SQLExceptions? (Choose all that apply.)

- A. A single query either executes correctly or throws a single exception
- B. A single query can throw many exceptions
- C. If a single query throws many exceptions, the exceptions can be captured by invoking SQLException.getExceptions(), which returns a List
- D. If a single query encounters more than one exception-worthy problem, a SQLException is created for each problem encountered
- E. If a single query throws many exceptions, the exceptions can be captured by iterating through the exceptions using SQLException.getNextException()

10 Which are true about the results of a query?

- A. The results are stored in a `ResultSet` object
- B. The results are stored in a `List` of type `Result`
- C. By default, the results remain synchronized with the database
- D. The results are accessed via iteration
- E. Once you have the results, you can retrieve a given row without needing to iterate
- F. The results are stored in a `List` of type `ResultSet`

## A SELF TEST ANSWERS

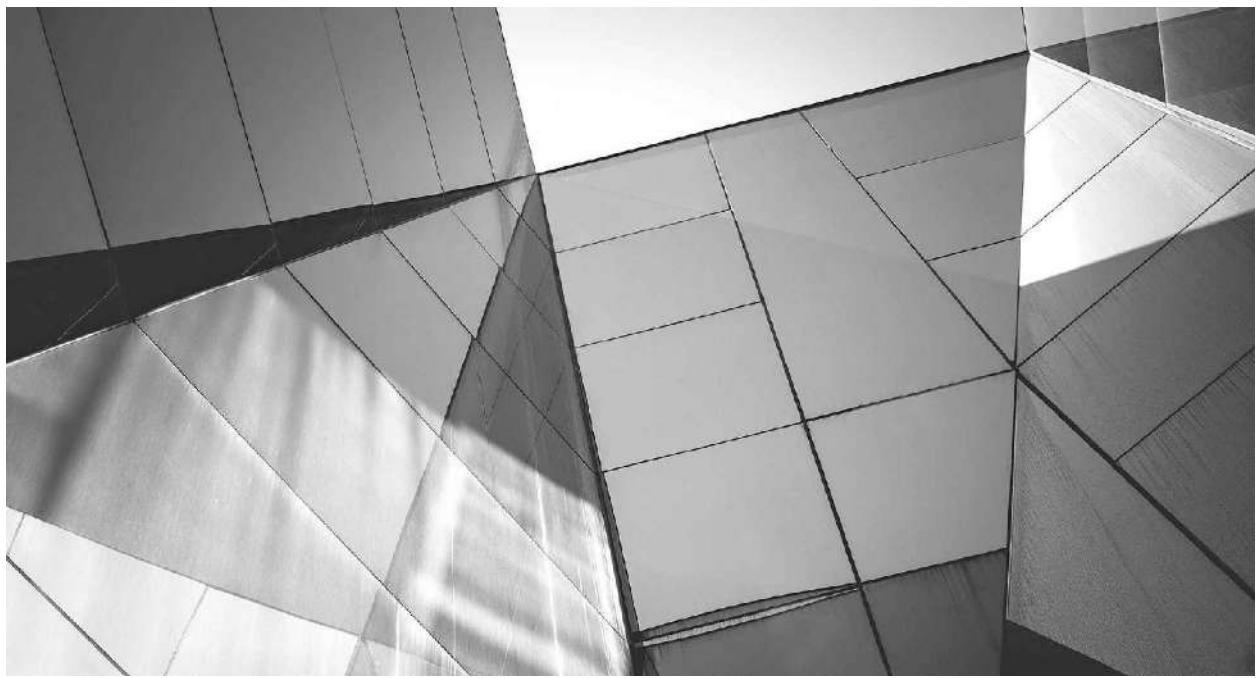
1.  **B** is correct. Prior to JDBC 4.0, JDBC drivers were required to register themselves with the `DriverManager` class by invoking `DriverManager.register(this);` after the driver was instantiated through a call from the classloader. The `Class.forName()` method calls the classloader, which, in turn, creates an instance of the class passed as a `String` to the method.  
 **A** is incorrect because this method is meant to be invoked with an instance of a `Driver` class. **C** is incorrect because `DatabaseMetaData` does not have a `loadDriver` method, and the purpose of `DatabaseMetaData` is to return information about a database connection. **D** is incorrect because, again, while the method sounds right, the arguments are not of the right types, and this method is actually the one called by `DriverManager.getConnection` to get a `Connection` object. **E** is incorrect because although this method returns a `Driver` instance, one has to be loaded and registered with the `DriverManager` first. (OCP Objective 11.2)
2.  **A, B, and E** are correct. To be JDBC 4.0 compliant, a JDBC driver must support the ability to autoload the driver by providing a file, `META-INF/services/java.sql.Driver`, that indicates the fully qualified class name of the `Driver` class that `DriverManager` should load on startup. The JDBC driver must implement the interfaces for `Driver`, `Connection`, `Statement`, `ResultSet`, and others. The driver must also support transactions.  
 **C** and **D** are incorrect. It is not a requirement to support scrollable or updatable `ResultSets`, although many drivers do. If, however, the driver reports that through `DatabaseMetaData` it supports scrollable and updatable

ResultSets, then the driver must support all of the methods associated with cursor movement and updates. **F** is incorrect. The JDBC requires that the driver support SQL92 entry-level grammar and the SQL command DROP TABLE (from SQL92 Transitional Level). **G** is incorrect. Although JDBC 4.0 drivers must support PreparedStatement, CallableStatement is optional and only required if the driver returns true for the method DatabaseMetaData.supportsStoredProcedures. (OCP Objective 11.2)

3.  **C, D, and F** are correct. DatabaseMetaData provides data about the database and the Connection object. The name, version, and other JDBC driver information are available, plus information about the database, including the names of stored procedures, functions, SQL keywords, and more. Finally, the default transaction isolation level and data about what transaction levels are supported are also available through DatabaseMetaData.
  - ☒ **A** and **B** are incorrect, as they are really about the result of a query with the database. Column count is available through a ResultSetMetaData object, but a row count requires that you, as the developer, move the cursor to the end of a result set and then evaluate the cursor position. **E** is incorrect. There is no method defined to return the last query in JDBC. (OCP Objective 11.1)
4.  **E** is correct. When the ResultSet returns, the cursor is pointing before the first row of the ResultSet. You must invoke the next() method to move to the next row of results *before* you can read any data from the columns. Trying to read a result using a getxxxx method will result in a SQLException when the cursor is before the first row or after the last row.
  - ☒ **A, B, D, and F** are incorrect based on the above. Note about **C**: the ResultSet returned from executeQuery will never be null. (OCP Objective 11.3)
5.  **D** is correct.
  - ☒ Note that answer **E** is close, but will not compile because the executeUpdate(query) method returns an integer result. **A** will compile correctly, but throw a SQLException at runtime—the executeQuery method cannot be used on INSERT, UPDATE, DELETE, or DDL SQL queries. **B** will not compile because the createStatement method does not take a String argument for the query. **C** is incorrect because Statement does not have a setQuery method and this fragment will not compile. (OCP

### Objective 11.3)

6.  **E** is correct. Recall that the `try-with-resources` statement on line X will automatically close the resource specified at the close of the `try` block (when the closing curly brace is reached) and closing the `Statement` object automatically closes any open `ResultSets` associated with the `Statement`. The `SQLException` thrown is that the `ResultSet` is not open. To fix this code, move the `while` statement into the `try-with-resources` block.  
 **A, B, C, D, and F** are incorrect based on the above. (OCP Objective 11.3)
7.  **B, C, D, and E** are correct.  
 **A** and **F** are incorrect. They are not interfaces required by JDBC. To query a database, the `Statement` interface is used, not the plausibly named, but mythical, `Query` interface. (OCP Objective 11.1)
8.  All of the answers are correct (**A, B, C, D, E**). `SELECT` statements will produce a `ResultSet` even if there are no rows. `INSERT`, `UPDATE`, and `DELETE` statements all produce an update count, even when the number of rows affected is 0. (OCP Objective 9.3)
9.  **B, D, and E** are correct.  
 **A** is incorrect because a single query can throw many exceptions. **C** is incorrect; when many exceptions are thrown, you must use `getNextException()` to iterate through them. (OCP Objective 11.3)
10.  **A, D, and E** are correct. For **E** you can use, for example, `getRow()`.  
 **B** is incorrect; results are stored in a `ResultSet` object. **C** is incorrect; once a `ResultSet` is created, it does NOT stay synchronized with the database unless the `ResultSet` was created with one of the `CONCUR_XXX` cursor types. **F** is incorrect based on the above. (OCP Objective 11.3)



# A

## About the Online Content

This book comes complete with Total Tester Online customizable practice exam software with 170 practice exam questions. In addition to preparing readers for the OCP Java SE 8 Programmer II exam, this book includes coverage of nearly all of the upgrade exam objectives for candidates recertifying from OCP Java SE 7 or OCP Java SE 6 credentials. You may obtain content covering the handful of upgrade objectives not in this book from the McGraw-Hill Professional Media Center.

## McGraw-Hill Professional Media Center Download

To access the supplemental upgrade objective content, visit McGraw-Hill Professional's Media Center by clicking the link below and entering this e-book's 13-digit ISBN and your e-mail address. You will then receive an e-mail message with a download link for the additional content.

<http://mhprofessional.com/mediacenter/>

This e-book's ISBN is 9781260117370.

Once you've received the e-mail message from McGraw-Hill Professional's Media Center, click the link included to download a zip file. Extract all of the files from the zip file and save them to your computer. If you do not receive the e-mail, be sure to check your spam folder.

## Total Tester Online System Requirements

We recommend and support the current and previous major versions of Chrome,

Firefox, Microsoft Edge, and Safari. These browsers update frequently, and sometimes an update may cause compatibility issues with the Total Tester Online or other content hosted on the Training Hub. If you run into a problem using one of these browsers, please try using another one until the problem is resolved.

## Single User License Terms and Conditions

Online access to the digital content included with this book is governed by the McGraw-Hill Education License Agreement outlined next. By using this digital content, you agree to the terms of that license.

**Access** To register and activate your Total Seminars Training Hub account and access your online practice exam, simply follow these easy steps:

1. Go to [hub.totalsem.com/mheclaim](http://hub.totalsem.com/mheclaim).
2. To register and create a new Training Hub account, enter your e-mail address, name, and password. No further information (such as credit card number) is required to create an account.
3. If you already have a Total Seminars Training Hub account, select Log In and enter your e-mail and password.
4. Enter your Product Key: **m2gw-4nj6-k3zd**
5. Click to accept the user license terms.
6. Click Register and Claim to create your account. You will be taken to the Training Hub and have access to the content for this book.

Duration of License Access to your online content through the Total Seminars Training Hub will expire one year from the date the publisher declares the book out of print.

Your purchase of this McGraw-Hill Education product, including its access code, through a retail store is subject to the refund policy of that store.

The Content is a copyrighted work of McGraw-Hill Education and McGraw-Hill Education reserves all rights in and to the Content. The Work is © 2018 by McGraw-Hill Education, LLC.

**Restrictions on Transfer** The user is receiving only a limited right to use the Content for user's own internal and personal use, dependent on purchase and

continued ownership of this book. The user may not reproduce, forward, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish, or sublicense the Content or in any way commingle the Content with other third-party content, without McGraw-Hill Education's consent.

**Limited Warranty** The McGraw-Hill Education Content is provided on an "as is" basis. Neither McGraw-Hill Education nor its licensors make any guarantees or warranties of any kind, either express or implied, including, but not limited to, implied warranties of merchantability or fitness for a particular purpose or use as to any McGraw-Hill Education Content or the information therein or any warranties as to the accuracy, completeness, currentness, or results to be obtained from, accessing or using the McGraw-Hill Education content, or any material referenced in such content or any information entered into licensee's product by users or other persons and/or any material available on or that can be accessed through the licensee's product (including via any hyperlink or otherwise) or as to non-infringement of third-party rights. Any warranties of any kind, whether express or implied, are disclaimed. Any material or data obtained through use of the McGraw-Hill Education content is at your own discretion and risk and user understands that it will be solely responsible for any resulting damage to its computer system or loss of data.

Neither McGraw-Hill Education nor its licensors shall be liable to any subscriber or to any user or anyone else for any inaccuracy, delay, interruption in service, error or omission, regardless of cause, or for any damage resulting therefrom.

In no event will McGraw-Hill Education or its licensors be liable for any indirect, special or consequential damages, including but not limited to, lost time, lost money, lost profits or good will, whether in contract, tort, strict liability or otherwise, and whether or not such damages are foreseen or unforeseen with respect to any use of the McGraw-Hill Education content.

## Total Tester Online

Total Tester Online access with this book provides you with a simulation of the Java SE 8 Programmer II (1Z0-809) exam. Exams can be taken in Practice Mode or Exam Mode. Practice Mode provides an assistance window with hints, references to the book, explanations of the correct and incorrect answers, and the option to check your answer as you take the test. Exam Mode provides a simulation of the actual exam. The number of questions, the types of questions,

and the time allowed are intended to be an accurate representation of the exam environment. The option to customize your quiz allows you to create custom exams from selected domains or chapters, and you can further customize the number of questions and time allowed.

To take a test, follow the instructions provided in the previous section to register and activate your Total Seminars Training Hub account. When you register, you will be taken to the Total Seminars Training Hub. From the Training Hub Home page, select your exam from the Study drop-down list at the top of the page or from the list of Products You Own on the Home page. You can then select the option to customize your quiz and begin testing yourself in Practice Mode or Exam Mode. All exams provide an overall grade and a grade broken down by domain.

## Technical Support

For questions regarding the Total Tester software or operation of the Training Hub, visit [www.totalsem.com](http://www.totalsem.com) or e-mail [support@totalsem.com](mailto:support@totalsem.com).

For questions regarding book content, e-mail [hep\\_customer-service@mheducation.com](mailto:hep_customer-service@mheducation.com). For customers outside the United States, e-mail [international\\_cs@mheducation.com](mailto:international_cs@mheducation.com).

# INDEX

*Please note that index links point to page beginnings from the print edition. Locations are approximate in e-readers, and you may need to page down one or more times after clicking a link to get to the indexed material.*

## A

- absolute() method, 855–857
- abstract classes, 5
  - constructors, 116, 119
  - creating, 8–9
  - implementation, 106
  - inner, 458, 460
  - vs. interfaces, 10–11
  - overview, 7–8
- abstract methods
  - inheritance, 75
  - overview, 31–35
  - subclass implementation, 88
- accept() method, 498, 504–505
- access and access modifiers, 16–18
  - classes, 3–8
  - constructors, 118
  - encapsulation, 71
  - inner classes, 453, 458
  - key points, 55, 57
  - lambda expression variables, 494–496
  - levels, 40
  - local variables, 29

- overloaded methods, [94](#)
- overridden methods, [88–89](#)
- private, [21–22](#)
- protected and default members, [22–27](#)
- public, [18–20](#)
- static methods and variables, [146](#)
- accumulators in streams, [567–570](#)
- AclFileAttributeView interface, [283](#)
- add() method
  - ArrayDeque, [390](#)
  - BlockingQueue, [738](#)
  - collections, [349](#)
  - lists, [351, 393](#)
  - sets, [378, 393](#)
- addAll() method, [737](#)
- addFirst() method, [390](#)
- afterLast() method, [856, 858](#)
- alive thread state, [650–651](#)
- allMatch() method, [576–578, 586](#)
- American National Standards Institute (ANSI), [843](#)
- and() method, [497, 511–512](#)
- andThen() method
  - Consumer, [507–508](#)
  - Function, [516](#)
  - Predicate, [511](#)
- angle brackets (<>) in generic code, [396, 398–399](#)
- anonymous inner classes
  - argument-defined, [466–468](#)
  - key points, [477](#)
  - overview, [461](#)
  - plain-old, flavor one, [461–464](#)
  - plain-old, flavor two, [464–465](#)
- ANSI (American National Standards Institute), [843](#)
- anyMatch() method, [576–578, 586](#)
- append() method, [252](#)
- apply() method, [514–517](#)

applyAsDouble() method

  DoubleBinaryOperator, 566–567

  Function, 517

  ToDoubleFunction, 561

applyAsInt() method, 517

argument-defined anonymous inner classes, 466–468

arguments

  assertions, 174–175

  final, 31

  overloaded methods, 94, 97

  overridden methods, 90

  vs. parameters, 35–36

  super constructors, 120

  variable argument lists, 35–36

ArrayList class, 738–739, 795

ArrayList class, 347, 355, 389–392

ArrayListException class, 734

ArrayList class, 376

  basics, 357–358

  in collection hierarchy, 347–348

  collections, 733

  description, 352

  element order, 350–351

  sorting, 363–365

ArrayListRunnable class, 734

arrays

  converting with lists, 375–376

  declaring, 43–45, 410–411

  key points, 60, 431

  methods, 392

  polymorphism, 408–410

  returning, 113

  searching, 373–375

  vs. streams, 543

  streams from, 548, 552

  type-safe, 394–395

Arrays.asList() method, 375, 392  
Arrays class, 347, 372–373  
ASCII set, 39  
asList() method, 375, 392  
AssertionError class, 172  
    appropriate use, 176  
    expression rules, 172  
assertions  
    appropriate use, 176–179  
    disabling, 174–175  
    expression rules, 172–173  
    key points, 193  
    overview, 170–172  
    running with, 174  
    working with, 174–176  
associative accumulators, 568–570  
associative pipeline operations, 772  
asterisks (\*)  
    globs, 294–296  
    SQL queries, 817  
atomic operations, 673  
atomic package, 722–723  
atomic variables, 722–725, 794  
AtomicInteger class, 725  
attributes  
    BasicFileAttributes, 283–285  
    common, 287  
    DosFileAttributes, 285–287  
    interface types, 282–283  
    key points, 316, 430  
    PosixFileAttributes, 286–287  
    reading and writing, 280–282  
autoboxing with collections, 358–362  
AutoCloseable interface, 878  
autocloseable resources  
    key points, 194

with try-with-resources statements, 185–189  
automatic local variables, 41–43  
Automatic Resource Management feature, 186  
automatic variables. *See* local variables  
available processors, 751  
average() method  
    associativity, 569  
    DoubleStream, 562–563, 565  
    optionals, 572  
    parallel streams, 772, 787  
    streams, 571  
averaging streams, 597–599  
averagingInt() method, 597–598, 603  
await() method  
    barriers, 745  
    conditions, 731

## B

backed collections, 385–387  
backslashes (\)  
    globbs, 294–296  
    properties files, 228  
barriers, concurrency, 741–745  
BaseStream interface, 770  
BasicFileAttributes interface, 282–285  
BasicFileAttributeView interface, 283  
beforeFirst() method, 856, 859  
between() method, 217  
BiConsumer interface  
    arguments, 499  
    methods and description, 523  
    working with, 504–505  
BiFunction interface  
    arguments, 515, 517

- methods and description, [523](#)
- BIGINT** data type, [847](#)
- BINARY LARGE OBJECT (BLOB)**, [843](#)
- BinaryOperator operators, [566](#), [787](#)
- binarySearch() method
  - arrays, [392](#)
  - collections, [392](#)
  - overview, [373](#)–[375](#)
- BiPredicate interface
  - methods and description, [523](#)
  - working with, [513](#)
- BLOB (BINARY LARGE OBJECT)**, [843](#)
- blocked thread state
  - considerations, [678](#)–[679](#)
  - deadlocks, [684](#)–[685](#)
  - description, [660](#)
- blocking queues, [737](#)–[741](#), [795](#)
- BlockingQueue collection, [737](#)–[738](#)
  - behavior, [738](#)–[739](#)
  - bounded queues, [739](#)
  - LinkedTransferQueue, [740](#)–[741](#)
  - special-purpose queues, [739](#)–[740](#)
- blocks
  - initialization, [137](#)–[139](#)
  - synchronized, [675](#)–[678](#)
  - synchronizing, [678](#)
- bookseller database overview, [819](#)–[822](#)
- boolean type and values
  - bit depth, [39](#)
  - SQL, [843](#), [847](#)
  - wrappers, [361](#)
- BooleanSupplier interface, [499](#)
- bounded queues, [739](#)
- braces ({}). *See* curly braces ({})
- BrokenBarrierException class, [745](#)
- buckets for hashcodes, [340](#)–[343](#)

BufferedReader class, [251](#)  
BufferedWriter class  
    description, [251](#)  
    methods, [259](#)  
    using, [258–259](#)  
bugs. *See* exceptions  
built-in functional interfaces, [497, 527](#)  
bytes  
    ranges, [39](#)  
    wrappers, [361](#)

## C

cached thread pools, [751](#)  
Calendar class, [207–208](#)  
call stacks with threads, [644–646, 651](#)  
Callable interface  
    overview, [753–754](#)  
    pipelines, [769](#)  
cancelRowUpdates() method, [862–864](#)  
canExecute() method, [282](#)  
canRead() method, [282](#)  
canWrite() method, [282](#)  
CAS (Compare And Swap) feature, [725](#)  
case sensitivity, SQL, [818](#)  
casts  
    with equals(), [337–338](#)  
    key points, [151](#)  
    overview, [101–104](#)  
catch clause. *See* try and catch feature  
categories for functional interfaces, [498](#)  
ceiling() method, [384](#)  
ceilingKey() method, [384](#)  
chaining  
    constructors, [117](#)

I/O classes, [258](#)  
changeable data, synchronizing, [680](#)  
CHARACTER LARGE OBJECT (CLOB), [843](#)  
characters and char type, [847](#)  
    bit depth, [39](#)  
    globs, [295–296](#)  
    wrappers, [361](#)  
checked exceptions  
    interface implementation, [106](#)  
    overloaded methods, [94](#)  
    overridden methods, [90, 92](#)  
ChronoUnit class, [207, 216–217](#)  
Class class, [677](#)  
Class.forName() method, [832–833](#)  
ClassCastException class  
    downcasts, [102](#)  
    with equals(), [337–338](#)  
classes  
    access, [3–8](#)  
    constructors. *See* constructors  
    dates and times, [224–226](#)  
    declaring, [3](#)  
    defining, [2–3](#)  
    extending, [4, 84](#)  
    final, [6–7, 45–46](#)  
    immutable, [134–137](#)  
    inner. *See* inner classes  
    interface implementation, [106](#)  
    literals, [677](#)  
    member, [450](#)  
    member declarations, [16](#)  
    names, [143, 175](#)  
    thread-safe, [681–684](#)  
clearWarnings() method, [875](#)  
CLOB (CHARACTER LARGE OBJECT), [843](#)  
close() method, [256, 875–879](#)

Closeable interface, 188–189  
closing SQL resources, 875–879  
code overview  
    coding to interfaces, 357–358  
    synchronizing, 668–673  
collect() method, 589–590  
    description, 601  
    stream reduction, 592–593  
    values from streams, 788–790  
Collection interface, 347–349, 770  
collections and Collections Framework, 44, 332  
    ArrayDeque class, 389–392  
    ArrayList basics, 357–358  
    autoboxing, 358–362  
    backed, 385–387  
    blocking queues, 737–741  
    Comparable interface, 365–367  
    Comparator interface, 367–368  
    concurrent, 733–741  
    converting arrays and lists, 375–376  
    copy-on-write, 735–736  
    diamond syntax, 362–363  
    hashcodes for, 340  
    implementation classes, 356  
    interfaces and classes, 347–351  
    key points, 429–430  
    legacy, 396–398  
    List interface, 351–352  
    lists, 376–378  
    Map interface, 353–354  
    maps, 379–382  
    methods, 392  
    mixing generic and nongeneric, 399–404  
    operations, 346–347  
    ordered, 349–351  
    overview, 346

polling, 384  
PriorityQueue class, 387–388, 393–394  
Queue interface, 354–355  
searching, 373–375  
searching TreeSets and TreeMaps, 382–383  
serialization, 313  
Set interface, 352–353  
sets, 378–379  
sorted, 351  
sorting, 363–372  
streams from, 546–547  
thread-safe, 736  
unboxing problems, 405  
working with, 733–734

Collections class, 347–349, 363  
Collections.synchronizedList() method, 682–684, 734  
Collector interface, 589–592  
Collectors class, 589–592  
collectors for streams, 600–603  
colons (:)

- assertions, 172
- URLs, 269

column indexes, 844  
combining I/O classes, 258–261  
command-line arguments for assertions, 175–176, 178  
comments, properties files, 227–229

Comparable interface

- vs. Comparator, 369
- concurrent collections, 737
- functional interfaces, 498
- lambda expressions, 369–372, 472–473
- sort orders, 351
- working with, 365–367

Comparator interface

- vs. Comparable, 369
- concurrent collections, 737

sort orders, 351  
working with, 367–368

Compare And Swap (CAS) feature, 725

compare() method, 368–370, 498

compareAndSet() method, 725

compareTo() method, 365–368, 581–582

comparing() method, 582–583

compiler and compiling

- casts, 102
- interface implementation, 105–106
- overloaded methods, 97
- warnings and fails, 401–404

compose() method, 516

compute() method

- ForkJoinTask, 757
- RecursiveAction, 761

computeIfAbsent() method, 515–516

concrete classes

- abstract methods implemented by, 88
- creating, 8–9
- subclasses, 32–33

concrete methods, 84

CONCUR\_READ\_ONLY cursor type, 852–853, 867

CONCUR\_UPDATABLE cursor type, 853–854, 862

concurrency, 645, 722

- atomic variables, 722–725
- collections, 733–741
- Cyclicbarrier, 741–745
- Executors. *See* Executors
- Fork/Join Framework. *See* Fork/Join Framework
- key points, 703, 794–798
- locks, 726–733
- parallel streams. *See* parallel streams
- ThreadPools. *See* ThreadPools

ConcurrentHashMap class, 736–737

ConcurrentLinkedDeque class, 736

ConcurrentLinkedQueue class, 736  
ConcurrentMap interface, 737  
ConcurrentSkipListMap class, 737  
ConcurrentSkipListSet class, 737  
Condition interface, 731–732  
conditions for locks, 730–732  
connect() method, 829  
Connection interface, 823–824, 875  
connections  
    databases, 816–817  
    DriverManager class, 825–830  
consistency in equals() contract, 339  
Console class  
    description, 252  
    working with, 265–266  
console() method, 265  
constant specific class body, 50–53  
constants  
    enum, 48  
    interface, 12–13  
constructing statements, 836–839  
constructors, 115  
    basics, 116  
    chaining, 117  
    declarations, 36–37  
    default, 118–120  
    enums, 50–51  
    inheritance, 75, 123  
    key points, 59, 152–153  
    overloaded, 123–128  
    rules, 118–119  
    singleton design pattern, 131  
    super() and this() calls, 126  
Consumer interface, 498, 504–508, 523  
contains() method  
    collections, 349

- lists, 393
- sets, 393
- containsKey() method, 393
- containsValue() method, 393
- contracts in JDBC, 823
- controls, 3
- conversions
  - arrays and lists, 375–376
  - return type, 114
  - strings to URIs, 269
  - types. *See* casts
- Coordinated Universal Time (UTC), 211–212
- copy() method, 272
- copy-on-write collections, 735–736
- copying files, 272–273
- CopyOnWriteArrayList collection, 735–736
- CopyOnWriteArraySet collection, 736
- cost reduction, object-oriented design for, 82
- count() method
  - return values, 571
  - stream elements, 543–544
  - stream reduction, 559, 565
  - stream values, 788
- counting
  - instances, 140
  - streams, 599
- counting() method, 599, 603
- covariant returns, 112–113
- CPU-intensive vs. I/O-intensive tasks, 748
- createNewFile() method, 254–255, 261–262, 270
- creationTime() method, 287
- credentials for database, 827
- CRUD operations, 817–818
- curly braces ({})
  - abstract methods, 32
  - anonymous inner classes, 462–463, 465

globbs, 295–296  
inner classes, 453, 458, 462–463  
lambda expressions, 493  
methods, 32  
currently running thread state, 658  
`currentThread()` method, 653  
cursor types for ResultSets, 853  
`Cyclicbarrier`, 741–745

## D

daemon threads, 646  
data types in ResultSets, 843  
`DatabaseMetaData` class, 868–873  
databases  
    connections, 816–817  
    credentials, 827  
    JDBC. *See* JDBC API  
    overview, 814–816  
`DataSource` class, 831  
`Date` class  
    description, 207–208  
    with SQL, 843  
`DATE` data type, 843, 847  
dates  
    adjustments, 213–214  
    classes, 224–226  
    durations, 216–217  
    examples, 219–220  
    format attributes, 281  
    formatting, 220–221  
    instants, 217–219  
    `java.time.*` classes, 208–210  
    key points, 238–239  
    locales, 222–224

overview, 207  
periods, 214–216  
zoned, 211–213

DateTimeFormatter class, 209–210, 220–222, 225

dead thread state, 651, 661

deadlocks

- key points, 704
- threads, 684–685
- tryLock() for, 728–730

Deadly Diamond of Death, 84

declarations

- arrays, 43–45, 410–411
- class members, 16
- classes, 3
- constructors, 36–37
- enum elements, 50–53
- enums, 48–50
- generics, 419–420
- interface constants, 12–13
- interfaces, 9–12
- reference variables, 40
- return types, 112–114
- variables, 37–47, 59–60

decoupling tasks from threads, 749–751

default access

- description, 3
- overview, 4–5
- and protected, 16, 22–27

default protection, 16

defaultReadObject() method, 308–309

defaults

- constructors, 118–120
- interface methods, 14
- locales, 234
- thread priorities, 666

defaultWriteObject() method, 309

defining

    classes, 2–3

    inner classes, 458

    threads, 647–648

Delayed interface, 740

DelayQueue class, 738–740, 795

delete() method, 263–264, 272

DELETE operation for SQL, 818

deleteIfExists() method, 273

deleteRow() method, 864–865

deleting files, 272–273

depth-first searches, 291

Deque interface, 389

descending order in collections, 384

descendingMap() method, 384

descendingSet() method, 384

deserialization process, 252

design patterns

    description, 129

    factory, 825–826

        singleton. *See* singleton design pattern

diamond syntax, 362–363

digits in globs, 295–296

directories

    creating, 270–271

    DirectoryStream, 288–289

    FileVisitor, 289–293

    iterating through, 288–289

    key points, 316–317

    renaming, 299

    working with, 261–265

DirectoryStream interface, 288–289

disabling assertions, 174–175

Disk Operating System (DOS), 283

DISTINCT characteristic for streams, 778

distinct() method, 584, 587, 777, 780

distributed-across-the-buckets hashcodes, 342  
divide and conquer technique, 756–757  
DOS (Disk Operating System), 283  
DosFileAttributes interface, 283, 285–287  
DosFileAttributeView interface, 283  
dots (.)  
    access, 17  
    class names, 143  
    instance references, 143  
    variable argument lists, 36  
double type  
    ranges, 39  
    SQL, 847  
DoubleBinaryOperator operators, 566–567  
DoubleConsumer interface, 501, 504  
DoubleFunction interface, 517  
DoublePredicate interface, 513  
DoubleStream interface  
    description, 551–552  
    methods, 570–571, 601  
    optionals, 572  
DoubleToIntFunction interface, 517  
DoubleToLongFunction interface, 517  
downcasts, 102–103  
Driver interface, 829, 832  
DriverManager class  
    database connections, 816  
    description, 825  
    key points, 882  
    overview, 826–828  
    registering JDBC drivers, 828–830  
drivers, JDBC, 824, 828–830, 832–833  
durations  
    dates, 216–217  
    threads, 657  
Durations class, 207, 216–217, 225

# E

eager initialization, [132](#)  
element existence, testing for, [577](#)–578  
element() method, [739](#)  
elevator property, [236](#)  
eligible thread state, [658](#)  
ellipses (...) in variable argument lists, [36](#)  
embarrassingly parallel problems, [764](#)–766, [770](#)–771  
empty() method, [576](#)  
empty optionals, [575](#)–576  
enabling assertions, [174](#)–175  
encapsulation  
    benefits, [82](#)  
    key points, [149](#)  
    overview, [70](#)–73  
ENTRY\_CREATE type, [298](#)–299  
ENTRY\_DELETE type, [298](#)–299  
ENTRY MODIFY type, [298](#)–299  
entrySet() method, [547](#)  
Enum class, [380](#)  
enums, [48](#)  
    constants, [48](#)  
    declaring, [48](#)–50  
    element declarations, [50](#)–53  
    key points, [60](#)–61  
equal signs (=)  
    reference equality, [334](#)–335  
    wrappers, [360](#)–361  
equality and equality operators  
    hashcodes, [344](#)–346  
    references, [334](#)–335  
equals() method, [332](#)  
    arrays, [392](#)  
Comparable, [498](#)

contract, 339  
description, 333  
implementing, 336–338  
key points, 428–429  
maps, 353, 379–382  
overriding, 334–335  
Set, 352  
wrappers, 360–361  
erasure, type, 403  
escape characters and sequences in globs, 295  
event handlers, 451–452  
ExceptionInInitializerError class, 139  
exceptions  
    interface implementation, 106  
    JDBC, 873–879  
    overridden methods, 90, 92  
    rethrowing, 182–184  
    suppressed, 190–192  
    try and catch. *See* try and catch feature  
    try-with-resources feature, 185–189, 194, 877–878  
exclamation points (!)  
    properties files, 227  
    wrappers, 360–361  
exclusive-OR (XOR) operator, 343  
execute() method  
    description, 839  
    overview, 837–838  
execute permission, 281–282, 286  
executeQuery() method  
    description, 839  
    overview, 836  
executeUpdate() method  
    description, 839  
    overview, 836–839  
ExecutionException class, 754  
Executor class, 749

Executors, 745–746

  Callable interface, 753–754

  CPU-intensive vs. I/O-intensive tasks, 748

  decoupling tasks from threads, 749–751

  ExecutorService shutdown, 754–755

  key points, 796

  parallel tasks, 746–747

  thread limits, 747

  thread pools, 751–752

  ThreadLocalRandom, 754

  turns, 748–749

  Executors.newCachedThreadPool() method, 752

  Executors.newFixedThreadPool() method, 752

  ExecutorService, 751–752

    Callable, 753–754

    shutdown, 754–755

  ExecutorService.shutdownNow() method, 755

  existence, stream elements, 577–578

  exists() method, 254–255, 270, 273

  exit() method, 755

  explicit values in constructors, 117

  expressions

    assertions, 172–173

    globs, 295–296

    regular, 296

  extended ASCII set, 39

  extending

    classes, 4, 84

    inheritance in, 76

    interfaces, 107

    Thread class, 647–648

  extends keyword

    illegal uses, 110

    IS-A relationships, 79

# F

factory design patterns, [825–826](#)  
fails vs. warnings, [402](#)  
FIFO (first-in, first-out) queues, [354–355](#)  
File class  
    creating files, [252–255](#)  
    description, [251](#)  
    files and directories, [261](#)  
    key points, [315–316](#)  
    methods, [259](#)  
File.list() method, [264](#)  
file:/ protocol, [269](#)  
FileInputStream class  
    methods, [259](#)  
    working with, [257–258](#)  
FileNotFoundException class, [181](#)  
FileOutputStream class  
    methods, [259](#)  
    working with, [257–258](#)  
FileOwnerAttributeView interface, [283](#)  
FileReader class  
    description, [251](#)  
    methods, [259](#)  
    working with, [255–257](#)  
files, [261–265](#)  
    attributes. *See* attributes  
    copying, moving, and deleting, [272–273](#)  
    creating, [252–255](#), [270–271](#)  
    key points, [315](#)  
    navigating, [250–252](#)  
    permissions, [281–282](#)  
    renaming, [299](#)  
    searching for, [264](#)  
    streams from, [549–551](#)

Files class, [267](#), [282](#)  
Files.delete() method, [272](#)  
Files.deleteIfExists() method, [273](#)  
Files.getLastModifiedTime() method, [281](#)  
Files.notExists() method, [271](#), [273](#)  
Files.walkFileTree() method, [289](#)–[290](#)  
FileSystems.getDefault() method, [293](#)  
 FileUtils class, [273](#)  
FileVisitor interface, [289](#)–[293](#)  
FileWriter class  
    description, [251](#)  
    methods, [259](#)  
    working with, [255](#)–[257](#)  
filter() method for streams  
    with collections, [547](#)  
    description, [544](#)–[545](#), [556](#)–[557](#)  
    pipelines, [777](#)  
    return values, [571](#)  
final arguments, [31](#)  
final classes, [6](#)–[7](#), [45](#)–[46](#)  
final constants, [12](#)–[13](#)  
final methods  
    nonaccess member modifiers, [30](#)–[31](#)  
    overriding, [90](#)  
final modifiers  
    inner classes, [458](#)–[460](#)  
    variables, [29](#), [45](#)  
finalize() method, [333](#)  
finally clauses  
    key points, [194](#)  
    with try and catch, [179](#)–[183](#)  
findAny() method  
    optionals, [572](#)  
    parallel streams, [782](#)–[783](#)  
    Stream, [576](#), [579](#)–[580](#), [586](#)–[587](#)  
findFirst() method

optionals, 572–573, 575  
Stream, 576, 579, 586–587  
FindMaxPositionRecursiveTask task, 764  
first-in, first-out (FIFO) queues, 354–355  
first() method, 856, 859  
firstDayOfNextYear() method, 214  
fixed thread pools, 751  
flatMap() method, 605–606  
flatMapToDouble() method, 606  
flatMapToInt() method, 606  
flatMapToLong() method, 606  
flexibility from object orientation, 70  
float type and floating-point numbers, 847  
    classes, 5–6  
    ranges, 39  
floor() method, 384  
floorKey() method, 384  
flush() method, 256  
for-each loops, 736  
forEach() method  
    Consumer, 505–506  
    key points, 526  
    parallel streams, 780–782  
    pipelines, 554–555  
    streams, 549–551  
forEachOrdered() method, 770, 780–782  
foreign keys, 821–822  
Fork/Join Framework, 755–756  
    divide and conquer technique, 756–757  
    embarrassingly parallel problems, 764–766  
    ForkJoinPool, 757  
    ForkJoinTask, 757–758  
    join(), 760–761  
    key points, 796–797  
    RecursiveAction, 761–762  
    RecursiveTask, 762–764

work stealing, 759–760  
fork() method, 757  
ForkJoinPool class, 757, 775–777, 783–786  
ForkJoinTask class, 757–758  
format() method, 221, 252, 851  
formatting  
    dates and times, 207, 220–221  
    reports, 851–852  
forName() method, 832–833  
fromMillis() method, 287  
Function interface, 499, 514–517, 523  
functional interfaces, 496–497  
    binary versions, 529  
    built-in, 497  
    categories, 498  
    Comparable, 369–370, 472–473, 498  
    description, 490, 497–498  
    functions, 499, 514–517  
    key points, 527–528  
    operators, 517–518  
    overview, 521–522  
    primitive versions, 528–529  
    suppliers, 498–503  
    UnaryOperator, 529  
    writing, 520–521  
Future class, 753–754

## G

generate() method, 607, 611  
generics, 332  
    classes, 420–424  
    declarations, 419–420  
    equals(), 334–340  
    hashCode(), 340–346

key points, 431–433  
legacy code, 398–399  
methods, 407–419, 424–426  
mixing with nongeneric, 399–404  
overview, 394–395  
polymorphism, 405–407  
`toString()`, 333–334

`get()` method  
    `HashTable`, 230  
    lists, 351, 378, 393  
    maps, 382, 393  
    optionals, 576  
    paths, 268–270  
    `Supplier`, 498, 500  
    unboxing problems, 405

`getAndIncrement()` method, 725

`getAsDouble()` method, 563–564, 574

`getAsInt()` method, 501, 574

`getAsLong()` method, 574

`getBoolean()` method, 845

`getBundle()` method, 232, 234–235

`getColor()` method, 585

`getColumnCount()` method, 848–849

`getColumnDisplaySize()` method, 850

`getColumnName()` method, 849

`getColumns()` method, 869–871

`getConnection()` method, 826–828, 832

`getDate()` method, 846

`getDayOfWeek()` method, 211

`getDefault()` method, 235, 293

`getDelay()` method, 740

`getDisplayCountry()` method, 223

`getDisplayLanguage()` method, 223

`getDouble()` method, 845

`getDriverName()` method, 869, 872

`getDriverVersion()` method, 869, 872

getEpoch() method, 219  
getErrorCode() method, 874  
getFileAttributeView() method, 287  
getFileName() method, 274  
getFloat() method, 845  
getId() method, 657  
getInt() method, 845  
getLastModifiedTime() method, 281  
getLong() method, 845  
getMessage() method, 873  
getMetaData() method, 868–869  
getName() method, 274, 652  
getNameCount() method, 274  
getNextException() method, 874  
getNextWarning() method, 875  
getObject() method, 234, 846, 852, 865  
getParent() method, 274  
getPathMatcher() method, 293  
getProcedures() method, 869, 871–872  
getProperty() method, 230  
getResultSet() method, 838–839  
getRoot() method, 274  
getRow() method, 857  
getRules() method, 213  
getSQLState() method, 873  
getState() method, 651  
getString() method, 846  
getSuppressed() method, 878  
getTableName() method, 850  
getters encapsulation, 71  
getTime() method, 846  
getTimestamp() method, 846  
getUpdateCount() method, 838–839  
getWarnings() method, 875  
glob, 289, 294–296  
graphs, object, 303–306

Greenwich Mean Time (GMT), 211–212  
grouping streams, 593–597  
groupingBy() method, 593–597, 602  
groups in PosixFileAttributes, 286–287  
guarantees with threads, 655–656, 665

## H

handle and declare pattern, 182  
hard-coding credentials, 827  
HAS-A relationships  
    key points, 149  
    overview, 78–82  
hashCode() method  
    contract, 344–346  
    generics, 332–333  
    HashSet, 353  
    implementing, 342–344  
    key points, 428–429  
    maps, 379–382  
    overriding, 340–343  
hashcodes overview, 340–343  
HashMap collection, 347  
    description, 354  
    hashcodes, 340  
HashSet collection, 347  
    description, 353  
    hashcodes, 340  
    ordering, 378  
Hashtable collection, 230, 347  
    description, 354  
    keys, 336  
    ordering, 350  
hasNext() method, 377  
headMap() method, 386

heads of queues, 390  
headSet() method, 386  
hiding implementation details, 71  
hierarchy in tree structures, 275  
higher-level classes, 301  
higher() method, 384  
higherKey() method, 384

# I

identifiers  
    Map, 353  
    threads, 657  
identity arguments for streams, 567  
identity() method, 516–517  
IllegalMonitorStateException class, 693, 729  
IllegalThreadStateException class, 657  
immutable classes  
    key points, 153  
    overview, 134–137  
immutable objects, thread safe, 736  
implementation details, hiding, 71  
implementers of interfaces, 464–465  
implementing interfaces  
    key points, 151  
    overview, 105–111  
implements keyword  
    illegal uses, 110  
    IS-A relationships, 79  
indexes  
    ArrayLists, 350  
    columns, 844  
    List, 351  
    searches, 373  
indexOf() method, 351, 378, 393

IndexOutOfBoundsException class, 682  
information about ResultSets, 848–850, 868–873  
inheritance  
    access modifiers, 17–18, 24–25  
    constructors, 123  
    event handlers, 452  
    evolution, 75–77  
    HAS-A relationships, 80–82  
    IS-A relationships, 78–79  
    key points, 149  
    multiple, 84, 110–111  
    overview, 74–75  
    serialization, 309–312  
inherited methods, overriding, 91  
initialization blocks, 137–139  
    inheritance, 75  
    key points, 154  
initialization of variables, 42  
injection attacks, 838  
inner classes, 3, 450–451  
    anonymous. *See* anonymous inner classes  
    defining, 458  
    instantiating, 454–455  
    key points, 476  
    lambda expressions, 469–473, 478  
    method-local, 458–460  
    modifiers, 458  
    objects, 455–456  
    overview, 451–453  
    referencing instances, 456–458  
    regular, 453–454  
    static, 468–469  
input/output (I/O), 249  
    combining classes, 258–261  
    Console class, 265–266  
    directories. *See* directories

files. *See* files  
key points, 315–317  
path creation, 268–270  
Runnable and Callable, 754  
serialization, 301–313  
WatchService, 297–300  
input-output intensive vs. CPU-intensive tasks, 748  
InputStream.read() method, 748  
INSERT operation, 817–818  
inserting rows, 866–868  
insertion points in searches, 373  
insertRow() method, 863, 868  
instance methods  
    inheritance, 75  
    overriding, 90  
    polymorphic, 86  
    references, 520  
instance variables  
    constructors, 117  
    inheritance, 75  
    overview, 40–41  
instances  
    counting, 140  
    initialization blocks, 138  
    references to, 143, 456–458  
instantiation, 152–153  
    eager and lazy, 132  
    inner classes, 454–455  
    static nested classes, 468–469  
    threads, 644–646, 649–651  
Instants class, 207, 225  
instants in dates, 217–219  
IntConsumer interface, 499, 504, 523  
integers and int data type  
    ranges, 39  
ResultSets, 847

- wrappers, [361](#)
- interfaces, [71](#)
  - vs. abstract classes, [10–11](#)
  - attributes, [282–283](#)
  - constants, [12–13](#)
  - constructors, [119](#)
  - declaring, [9–12](#)
  - extending, [107](#)
  - functional. *See* functional interfaces
  - implementing, [105–111](#), [151](#)
  - JDBC, [823–824](#)
  - key points, [56](#)
  - methods, [14–15](#)
- intermediate operations
  - pipelines, [553](#)
  - streams, [544](#), [552–553](#)
- interrupt() method, [755](#)
- InterruptedException class
  - barriers, [745](#)
  - Callable, [754](#)
  - Conditions, [730–732](#)
  - sleep(), [662](#)
  - WatchService, [299](#)
- IntFunction interface
  - arguments, [517](#)
  - methods and description, [523](#)
- IntPredicate interface
  - arguments, [513](#)
  - methods and description, [523](#)
- IntStream interface, [551](#)
  - methods, [552](#), [571](#), [601](#)
  - optionals, [572](#)
- IntSupplier interface, [523](#)
- IntToDoubleFunction interface, [517](#)
- IntToLongFunction interface, [517](#)
- invokeAll() method, [762](#)

invoking

- overloaded methods, 96–98

- Polymorphic methods, 85

I/O. *See* input/output (I/O)

IOException class

- directories, 262

- files, 181

IS-A relationships

- key points, 149

- overview, 78–79

- polymorphism, 83

- return types, 115

- Serializable, 309

isAfterLast() method, 859–860

isAlive() method, 651

isBeforeFirst() method, 859

isDaylightSavings() method, 213

isDirectory() method, 284

isEqual() method, 497, 512–513

isFirst() method, 860

isHidden() method, 287

isInterrupted() method, 755

isLast() method, 860

isLeapYear() method, 220

ISO Latin-1 characters, 39

isParallel() method, 769–770

isPresent() method, 573–574, 576

isReadOnly() method, 287

Iterable class, 274

iterate() method, 607–608, 611

iteration

- collections, 350, 735–736

- directories, 288–289

- paths, 274–275

iterator() method

- collections, 349, 735–736

lists, 393

sets, 393

## Iterators

collections, 735–736

lists, 376–377

# J

java.io.Console class

description, 252

working with, 265–266

java.io.IOException class

directories, 262

files, 181

java.io.ObjectInputStream class

description, 252

working with, 258, 301–303

java.io.ObjectOutputStream class

description, 252

working with, 258, 301–303

java.io package

classes, 258–259

files and directories, 261–265

java.lang.Class class, 677

java.lang.ClassCastException class

downcasts, 102

with equals(), 337–338

java.lang.Enum class, 380

java.lang.Object class

collections, 396, 405

description, 332

equals() method. *See* equals() method

inheritance from, 74

threads, 659, 699

java.lang.Runnable interface

- executing, 746
- I/O activities, 754
- implementing, 648
- threads, 647, 699
- `java.lang.Runtime` class, 751
- `java.lang.StringBuilder` class, 681
- `java.lang.System` class, 226
- `java.lang.Thread` class
  - description, 644
  - extending, 647–648
  - methods, 646, 658
  - thread methods, 699
- Java Naming and Directory Interface (JNDI) lookup, 831
- `java.nio.file` package, 267–268, 288
- `java.nio.file.attribute` package, 267, 280
- `java.nio.file.Path` interface
  - key points, 315–316
  - methods, 274–275
  - working with, 267–268
- Java resource bundles, 233–234
- `java.sql.Connection` interface, 823–824, 875
- `java.sql.Date` class
  - description, 207–208
  - with SQL, 843
- `java.sql.Driver` interface, 829, 832
- `java.sql.DriverManager` class
  - database connections, 816
  - description, 825
  - key points, 882–883
  - overview, 826–828
  - registering JDBC drivers, 828–830
- `java.sql` package, 823
- `java.sql.ResultSet` interface, 823–824
- `java.time.*` classes, 208–210
- `java.time.Durations` class, 207, 216–217, 225
- `java.time.format.DateTimeFormatter` class, 209–210, 220–222, 225

`java.time.Instants` class, 207, 225  
`java.time.LocalDate` class, 207, 225–226  
`java.time.LocalDateTime` class, 207–210, 225–226  
`java.time.LocalTime` class, 207, 210, 225–226  
`java.time.OffsetDateTime` class, 207, 214, 225  
`java.time.Periods` class, 207, 225  
`java.time.temporal.ChronoUnit` class, 207, 216–217  
`java.time.temporal.TemporalAdjusters` class, 207, 214  
`java.time.ZonedDateTime` class, 207, 212–214, 218, 226  
`java.util.ArrayList` class. *See* `ArrayList` class  
`java.util.Calendar` class, 207–208  
`java.util.Collection` interface, 347–349  
`java.util.Collections` class, 347–349, 363  
`java.util.concurrent.atomic` package, 722–723  
`java.util.concurrent.Callable` interface  
    overview, 753–754  
    pipelines, 769  
`java.util.concurrent` collections, key points, 794–796  
`java.util.concurrent.Delayed` interface, 740  
`java.util.concurrent.Executor` class, 749  
`java.util.concurrent.ForkJoinPool` class, 757, 775–777, 783–786  
`java.util.concurrent.ForkJoinTask` class, 757–758  
`java.util.concurrent.Future` class, 753–754  
`java.util.concurrent.locks.Condition` interface, 731–732  
`java.util.concurrent.locks.Lock` interface, 727  
`java.util.concurrent.locks` package, 722–723, 726–727  
`java.util.concurrent.locks.ReentrantLock` class, 727–730  
`java.util.concurrent` package, 722. *See also* concurrency  
`java.util.concurrent.ThreadLocalRandom` class, 754  
`java.util.concurrent.ThreadPoolExecutor`, 752  
`java.util.Date` class  
    limitations, 207–208  
    with SQL, 843  
`java.util.function.Consumer` interface, 498, 504–508  
`java.util.function.Function` interface, 499, 514–517  
`java.util.function` package, 497, 521–523

java.util.function.Predicate interface, 369, 497–498, 508–514  
java.util.function.Supplier interface, 498–503  
java.util.Hashtable class, 230  
java.util.List interface, 347  
    implementations, 351–352  
    methods, 393  
    threads, 733  
java.util.Locale class  
    resource bundles, 231  
    working with, 222–223, 225  
java.util.logging.Logger class, 501–503  
java.util.NavigableMap interface, 347, 382  
java.util.NavigableSet interface, 347, 382  
java.util.NoSuchElementException class, 391  
java.util package, 385, 431  
java.util.Properties class, 226–230  
java.util.ResourceBundle class, 227, 231–233  
javax.sql.DataSource class, 831  
JDBC API, 814  
    database connections, 816–817  
    database overview, 814–816  
    driver implementation versions, 832–833  
    DriverManager class, 825–830  
    drivers, 824, 828–830  
    exceptions and warnings, 873–879  
    interfaces, 823–824  
    key points, 882  
    query submissions, 833–839  
    result sets. *See* ResultSets  
    SQL queries, 817–818  
    statements, 835–839  
    test database, 819–822  
    URL, 830–832  
JIT (Just In Time) compiler, 724  
JNDI (Java Naming and Directory Interface) lookup, 831  
join() method

Fork/Join Framework, 757, 760–761  
key points, 703  
locks, 679  
threads, 658, 666–668, 699  
join tables, 821–822  
joining() method, 599–600, 603  
joining streams, 599–600  
Just In Time (JIT) compiler, 724

## K

key.pollEvents() method, 299  
key.reset() method, 299  
keys  
databases, 815, 820–822  
hashtables, 336  
Map, 353  
properties files, 227  
keySet() method, 393

## L

Labels\_en.properties file, 231–232  
Labels\_fr.properties file, 231–232  
lambda expressions  
Comparable interface, 369–372, 472–473  
functional interfaces. *See* functional interfaces  
inner classes, 469–473, 478  
key points, 526  
method references, 518–520  
passing to methods, 494  
side effects, 506–507  
syntax, 490–493  
variable access from, 494–496  
LARGE OBJECT types, 843

large tasks, divide and conquer technique for, 756–757  
last() method, 856, 859  
lastDayOfMonth() method, 214  
lastModified() method, 282  
lastModifiedTime() method, 287  
lazy initialization, 132  
lazy streams, 555–556  
leap years, 220  
legacy code in generics, 398–399  
LIKE operator, 841  
limit() method, 607–608, 611, 777, 780  
lines() method, 549, 592–593  
LinkedBlockingDeque class, 738–739, 795  
LinkedBlockingQueue class, 738–739, 795  
LinkedHashMap class, 347, 354  
LinkedHashSet class, 347, 350, 353  
LinkedList class, 347, 352, 376  
LinkedTransferQueue class, 738, 740–741, 795  
list.iterator() method, 735–736  
list() method, 230, 264  
ListIterator interface, 736  
ListResourceBundle class, 233–234  
lists and List interface, 347  
    ArrayLists. *See* ArrayList class  
    collecting items in, 593  
    converting with arrays, 375–376  
    description, 349  
    implementations, 351–352  
    key points, 431  
    methods, 393  
    threads, 733  
    working with, 376–378  
literals  
    class, 677  
    globs, 295  
livelocks

- key points, 704
- threads, 685–686
- load() method, 230
- local variables
  - access modifiers, 29
  - inner classes, 459
  - key points, 58
  - on stack, 41
  - working with, 41–43
- LocalDate class, 207, 225–226
- LocalDateTime class, 207–210, 225–226
- Locale class
  - resource bundles, 231
  - working with, 222–223, 225
- locales
  - dates and times, 222–224
  - default, 234
  - key points, 239
  - resource bundles. *See* resource bundles
- localhosts in URLs, 831
- LocalTime class, 207–210, 225–226
- Lock interface, 727
- locks, 726–727
  - conditions, 730–732
  - obtaining, 678–679
  - ReentrantReadWriteLock, 732–733
  - synchronization, 674–677
- locks package, 722–723, 726–727
- log() method, 501–503
- Logger class, 501–503
- long type, 39
- LongConsumer interface, 501, 504
- LongFunction interface, 517
- LongPredicate interface, 513
- LongStream interface, 551–552
  - methods, 571, 601

optionals, [572](#)  
LongToDoubleFunction interface, [517](#)  
LongToIntFunction interface, [517](#)  
LONGVARCHAR data type, [847](#)  
loop constructs, wait() in, [697–699](#)  
lower-level classes, [301–302](#)  
lower() method, [384](#)  
lowercase characters in natural ordering, [394](#)  
lowerKey() method, [384](#)

## M

main() method  
    overloaded, [98](#)  
    threads, [644–646](#)  
maintainability, object orientation for, [70](#)  
map-filter-reduce operations  
    overview, [556–559](#)  
    streams, [570–572](#), [619–620](#)  
    working with, [560–564](#)  
map() method, [556–557](#), [570](#), [777](#)  
mapping() method, [595–596](#), [602](#)  
maps and Map interface, [347](#)  
    description, [349](#)  
    methods, [393](#)  
    overview, [353–354](#)  
    working with, [379–382](#)  
mapToDouble() method, [561–562](#), [570](#)  
mapToInt() method, [570](#)  
mapToLong() method, [570](#)  
mapToObj() method, [570](#)  
matching methods, [577–580](#)  
max() method  
    optionals, [572](#)  
    streams, [565](#), [571](#)

maxBy() method, 600, 603  
member modifiers, nonaccess, 29–36  
members  
    access. *See* access and access modifiers  
    declaring, 16  
    key points, 58–59  
META-INF/services/java.sql.Driver file, 832  
metadata for ResultSets, 848–850, 868–873  
method-local inner classes, 458–459  
    key points, 476–477  
    working with, 459–460  
methods  
    abstract, 31–35  
    access modifiers. *See* access and access modifiers  
    anonymous inner classes, 462  
    assertions, 176–178  
    enums, 50–51  
    final, 30–31, 45  
    generics, 407–419, 424–426  
    instance, 86, 90  
    interface implementation, 105–106  
    interfaces, 14–15  
    lambda expression references, 518–520  
    overloaded, 94–100  
    overridden, 87–93  
    parameters, 278  
    passing lambda expressions to, 494  
    static, 47, 139–142  
    synchronized, 35  
    variable argument lists, 35–36  
min() method  
    optionals, 572  
    streams, 565, 571  
minBy() method, 600, 603  
minusSeconds() method, 214  
minusWeeks() method, 214

minusX() method, [221](#)  
MissingResourceException class, [235](#)  
mixing generic and nongeneric collections, [399–404](#)  
mkdir() method, [262](#)  
modifiers. *See* access and access modifiers  
monitors for locking, [675, 726](#)  
move() method, [272](#)  
moveToCurrentRow() method, [863, 868](#)  
moveToInsertRow() method, [863, 867](#)  
moving files, [272–273](#)  
moving in ResultSets, [841–842, 852–861](#)  
multi-catch clauses, [179–183, 194](#)  
multidimensional arrays, [44](#)  
multiple inheritance, [84, 110–111](#)  
multiple threads, starting and running, [654–657](#)  
multithreading, [644](#)  
mutable data, synchronizing, [680](#)  
mutators in encapsulation, [71](#)

# N

names

- classes and interfaces, [175](#)
- constructors, [37](#), [116](#), [118](#)
- dot operator, [143](#)
- files and directories, [264](#), [299](#)
- threads, [652](#)
- native threads, [645](#)
- natural ordering, [394](#)
- NavigableMap interface, [347](#), [382](#)
- NavigableSet interface, [347](#), [382](#)
- navigation
  - ResultSets, [841](#)–[842](#), [852](#)–[861](#)
  - TreeSets and TreeMaps, [382](#)–[384](#)
- negate() method, [497](#), [511](#)–[512](#)
- negative numbers, representing, [38](#)
- nested classes, [3](#)
  - inner. *See* inner classes
  - static, [450](#), [468](#)–[469](#)
- new keyword for inner classes, [454](#)
- new thread state
  - description, [659](#)
  - starting threads, [651](#)
- newCachedThreadPool() method, [752](#)
- newFixedThreadPool() method, [752](#)
- newLine() method, [251](#)
- next() method
  - Iterator, [377](#)
  - ResultSets, [841](#)–[842](#), [852](#), [856](#)
  - String, [607](#)
- nextInt() method, [754](#)
- NIO.2
  - attributes, [280](#)–[287](#)

DirectoryStream, 288–289  
files and directories, 270–280  
files and paths, 267–268  
FileVisitor, 289–293  
key points, 315–316  
PathMatcher, 293–297  
permissions, 281–282  
WatchService, 297–300  
no-arg constructors, 118–119  
non-static fields, synchronizing, 680–681  
non-synchronized method, 675  
nonaccess member modifiers, 5–6, 29–30  
    abstract methods, 31–35  
    final arguments, 31  
    final methods, 30–31  
    key points, 55  
    methods with variable argument lists, 35–36  
    synchronized methods, 35  
noneMatch() method, 576–578, 586–587  
nongeneric code, updating to generic, 398–399  
nongeneric collections, mixing with generic, 399–404  
normalize() method, 276–277  
normalizing paths, 275–277  
NoSuchElementException class, 391, 563  
NoSuchFileNotFoundException class, 272  
notExists() method, 271, 273  
notify() method  
    Class, 699  
    description, 333  
    key points, 704  
    locks, 679  
    Object, 753  
    threads, 659, 690–692, 695, 697–699  
notifyAll() method  
    Class, 699  
    description, 333

Object, 753  
threads, 659, 690, 695–699  
now() method, 208, 218  
null values  
    returning, 113  
    wrappers, 362  
NullPointerException class, 362  
number signs (#) in properties files, 227  
NumberFormat class, 222  
numbers  
    primitives. *See* primitives  
    random, 754

## O

ObjDoubleConsumer interface, 504  
Object class  
    collections, 396, 405  
    description, 332  
    inheritance from, 74  
    threads, 659, 699  
object graphs, 303–306  
Object.notify() method, 753  
Object.notifyAll() method, 753  
object orientation (OO), 69–70  
    benefits, 82  
    casting, 101–104  
    constructors. *See* constructors  
    encapsulation, 70–73  
    inheritance, 74–82  
    initialization blocks, 137–139  
    interface implementation, 105–111  
    overloaded methods, 94–100  
    overridden methods, 87–93  
    polymorphism, 83–87

- return types, 112–114
- singleton design pattern, 128–133
- statics, 139–146
- Object.wait() method, 753
- ObjectInputStream class
  - description, 252
  - working with, 258, 301–303
- ObjectOutputStream class
  - description, 252
  - working with, 258, 301–303
- objects and object references
  - arrays, 44
  - inner classes, 455–456
  - overloaded methods, 96–97
  - updating columns with, 865–866
- ObjIntConsumer interface, 504
- ObjLongConsumer interface, 504
- of() method
  - dates, 209–210, 215, 221
  - optionals, 576
  - Stream, 548
- orElse() method, 576
- offer() method
  - ArrayQueue, 390
  - BlockingQueue, 738
  - LinkedList, 352
  - PriorityQueue, 387–388, 393
- offerFirst() method, 390
- OffsetDateTime class, 207, 214, 225
- ofMinutes() method, 217
- ofNullable() method, 575–576
- operating system thread limits, 747
- operations for streams, 556–559
- operators for functional interfaces, 517–518
- Optional class, 573
- OptionalDouble class, 562–564, 573

OptionalInt class, 573  
OptionalLong class, 573  
optionals, 562–564  
    key points, 620  
    working with, 572–576  
or() method, 497, 511–512  
order  
    ArrayList elements, 350–351  
    collections, 384  
    instance initialization blocks, 138  
    natural, 394  
    threads, 658  
ORDERED characteristic for streams, 778  
ordered collections, 349–351  
OutputStream.write() method, 748  
overloaded constructors, 123–128  
overloaded methods, 94–95  
    invoking, 96–98  
    key points, 150–151  
    legal, 95, 99  
    main(), 98  
    vs. overridden, 95, 100  
    polymorphism, 98  
    return types, 112  
overridden methods, 87–91  
    illegal, 92–93  
    invoking superclass, 91–92  
    vs. overloaded, 95, 100  
    @Override annotation, 93–94  
    polymorphism, 98  
    return types, 112–113  
    static, 146  
@override annotation, 93–94, 151  
overriding  
    anonymous inner class methods, 462  
    equals(), 334–335

hashcode(), 340–343  
key points, 150–151, 428–429  
private methods, 21–22  
run(), 647

## P

package-centric languages, 3  
package-level access, 4–5  
packages  
    access, 3  
    assertions, 175  
parallel Fork/Join Framework. *See* Fork/Join Framework  
parallel() method, 612–613, 767, 769–770, 779  
parallel streams, 612–614, 766–767  
    associative pipeline operations, 772  
    embarrassingly parallel problems, 770–771  
    findAny(), 782–783  
    forEach() and forEachOrdered(), 780–782  
    key points, 623, 796–798  
    pipelines, 767–770  
    RecursiveTask, 783–786  
    reduce(), 786–790  
    side effects, 773–774  
    stateful operations, 773–778  
    stateless operations, 773  
    unordered, 778–780  
parallel tasks, identifying, 746–747  
parallelStream() method, 769–770  
parameterized types, 396–397  
parameters  
    vs. arguments, 35–36  
    lambda expressions, 491–492  
    methods, 278  
    multi-catch and catch, 181–182

parentheses ()  
arguments, 31, 35  
lambda expressions, 493  
try-with-resources, 877–878  
parse() method, 209–210, 221  
parsing searches. *See* searches  
partitioning streams, 593–597  
partitioningBy() method, 596, 602  
passing lambda expressions to methods, 494  
passwords, hard-coding, 827  
PathMatcher, 293–297  
paths and Path interface  
    creating, 268–270  
    iterating through, 274–275  
    key points, 315–316  
    methods, 274–275  
    normalizing, 275–277  
    relativizing, 278–280  
    resolving, 277–278  
    retrieving, 273–275  
    working with, 266–268  
Paths class, 267, 315–316  
Paths.get() method, 268–270  
patterns, design  
    factory, 825–826  
    singleton. *See* singleton design pattern  
peek() method  
    ArrayDeque, 390  
    BlockingQueue, 739  
    LinkedList, 352  
    parallel streams, 788  
    pipelines, 767  
    PriorityQueue, 387–388, 393  
    streams, 559, 580  
percent signs (%) for LIKE operator, 841  
Periods class, 207, 225

periods in dates, 214–216  
permissions  
  files, 281–282  
  PosixFileAttributes, 286–287  
pipe () characters for multi-catch clauses, 180  
pipelines and pipeline operations  
  associative, 772  
  key points, 618  
  parallel streams, 767–770  
  stateful operations, 773–778  
  streams, 553–554  
plus() method, 217  
plusMinutes() method, 214, 217  
plusX() method, 221  
plusYears() method, 214  
poll() method  
  ArrayDeque, 390  
  BlockingQueue, 739  
  LinkedList, 352  
  PriorityQueue, 387–388, 393  
  WatchService, 299  
pollEvents() method, 299  
pollFirst() method, 384  
pollFirstEntry() method, 384  
polling, 384  
pollLast() method, 384, 391, 393–394  
polymorphism  
  abstract classes, 8  
  anonymous inner classes, 463  
  arrays, 408–410  
  generics, 405–407  
  inheritance, 77  
  key points, 149–150  
  overloaded and overridden methods, 98  
  overview, 83–87  
pools, thread. *See* ThreadPools

pop() method, 390–391, 393  
Portable Operating System Interface (POSIX), 283  
PosixFileAttributes interface, 283  
PosixFileAttributeView interface, 283, 286–287  
postVisitDirectory() method, 291  
predicates and Predicate interface  
    description, 497–498  
    functional interfaces, 498, 508–514  
    lambda expressions, 369  
    methods and description, 523  
    working with, 508–514  
preemptive multitasking, 749  
preemptive priority-based scheduling, 664–668  
preventing thread execution, 661  
previous() method, 856, 858  
preVisitDirectory() method, 291  
primary database keys, 815, 820–822  
primitives  
    arrays, 44  
    declarations, 37–39  
    final, 45  
    returning, 114  
    streams, 551  
    wrapper classes, 358–362  
printf() method, 252  
printing  
    file permissions, 282  
    reports, 850–852  
PrintInputStream class, 252  
println() method, 258, 519–520  
PrintWriter class, 252, 258–259  
priority of threads, 658, 664–668  
PriorityBlockingQueue class, 738, 795  
PriorityQueue class, 347, 355, 387–388, 393–394  
private modifiers, 16  
    inner classes, 458

overriding, 21–22  
overview, 21–22  
processors, available, 751  
Properties class, 226–230  
properties files, 226–230, 239  
protected modifiers, 16  
    inner classes, 458  
    working with, 22–27  
public access, 5  
public modifiers, 16  
    constants, 12–13  
    encapsulation, 71  
    inner classes, 458  
    overview, 18–20  
push() method, 390  
put() method, 230  
    BlockingQueue, 738  
    maps, 393  
putIfAbsent() method, 737

## Q

queries  
    key points, 882–883  
    result sets. *See* ResultSets  
    SQL, 817–818  
    submitting, 833–839  
query strings, 838  
question marks (?)  
    generics, 426  
    globs, 295–296  
Queue interface, 347, 354–355  
queues  
    blocking, 737–741  
    bounded, 739

description, 349  
LinkedTransferQueue, 740–741  
special-purpose, 739–740  
threads, 658  
quotes (', ") in SQL, 818

## R

race conditions, 672–673, 686–690  
random numbers, 754  
RandomInitRecursiveAction task, 764  
range() method, 607, 611  
ranges  
    numbers, 39  
    streams, 609, 611  
RDBMSs (Relational Database Management Systems), 815  
read() method, 251, 256, 748  
read-only objects, 736  
read permissions, 281–282, 286  
readAttributes() method, 286–287  
reading  
    attributes, 280–282  
    ResultSets, 842–847  
readLine() method, 251, 260, 263  
readObject() method, 302, 306–309  
readPassword() method, 265  
REAL data type, 847  
RecursiveAction class, 761–762  
RecursiveTask task, 762–764, 783–786  
Red-Black tree structure, 353  
redefined static methods, 146  
reduce operations and reduce() method  
    optionals, 572  
    parallel streams, 786–790  
    streams, 556–557, 571, 777

working with, 565–568  
ReentrantLock class, 727–730  
ReentrantReadWriteLock object, 732–733  
references, 83  
    casting, 101–104, 151  
    declaring, 40  
    equality, 334–335  
    inner classes, 456–458  
    instances, 143  
    lambda expression methods, 518–520  
    overloaded methods, 96–97  
    returning, 113  
reflexivity with equals(), 339  
registering JDBC drivers, 828–830  
regular expressions (regex) vs. globs, 296  
regular inner classes, 453–454  
Relational Database Management Systems (RDBMSs), 815  
relative() method, 856–858  
relative paths, 269  
relativize() method, 279–280  
relativizing paths, 278–280  
releasing locks, 675  
remove() method  
    BlockingQueue, 738  
    collections, 349, 737  
    description, 393–394  
    lists, 393  
    maps, 393  
    sets, 393  
removeFirst() method, 682  
removeIf() method, 511  
removeLast() method, 391, 394  
renameTo() method, 264  
renaming files and directories, 299  
replace() method, 737  
replaceAll() method, 515–516

reports, printing, 850–852  
reset() method, 299  
    barriers, 745  
    WatchService, 299  
resolve() method, 278  
resolving paths, 277–278  
resource bundles, 227, 231–233  
    default locales, 234  
    Java, 233–234  
    key points, 239  
    selecting, 235–237  
ResourceBundle class, 231–233  
resources  
    autocloseable, 185–189  
    closing, 875–879  
results, database, 817–818. *See also* ResultSets  
ResultSet interface, 823–824  
ResultSetMetaData class, 848  
ResultSets  
    cursor types, 853  
    information about, 848–850, 868–873  
    key points, 882–883  
    moving in, 841–842, 852–861  
    overview, 840  
    reading from, 842–847  
    reports, 850–852  
    updating, 861–862  
resume() method, 660  
rethrowing exceptions, 182–184  
retrieving path information, 273–275  
return type  
    constructors, 37, 116, 118  
    declarations, 112–114  
    key points, 151–152  
    lambda expressions, 492  
    overloaded methods, 94, 112

- overridden methods, [90](#), [112](#)–[113](#)
- returning values, [113](#)–[114](#)
- reuse, inheritance for, [76](#)
- reuseless code, [450](#)
- reverse() method, [392](#)
- reversed() method, [582](#)–[583](#)
- reverseOrder() method, [392](#)
- ride.in property, [236](#)
- rowDeleted() method, [865](#)
- rows
  - database, [814](#)–[815](#)
  - inserting, [866](#)–[868](#)
- rowUpdated() method, [863](#)
- rs.getObject() method, [852](#)
- rules
  - constructors, [118](#)–[119](#)
  - expression, [172](#)–[173](#)
- run() method
  - overriding, [647](#)
  - Runnable, [699](#)
  - threads, [646](#), [651](#)–[652](#), [656](#)
- Runnable interface
  - executing, [746](#)
  - I/O activities, [754](#)
  - implementing, [648](#)
  - threads, [647](#), [699](#)
- runnable thread state, [659](#)
- running thread state
  - description, [659](#)–[660](#)
  - threads, [659](#)
- running with assertions, [174](#)
- runtime
  - disabling assertions at, [174](#)
  - enabling assertions at, [174](#)
- Runtime class, [751](#)
- runtime exceptions for overridden methods, [90](#)

# S

scheduled thread pools, [752](#)  
`ScheduledThreadPoolExecutor`, [752](#)  
scheduler, thread, [658](#)  
schema, database, [819](#)–[822](#)  
scope of lambda expressions, [495](#), [506](#)  
searches  
    arrays and collections, [373](#)–[375](#)  
    files, [264](#)  
    key points, [431](#), [621](#)–[622](#)  
    with streams, [576](#)–[580](#), [586](#)–[588](#)  
    TreeSets and TreeMaps, [382](#)–[383](#)  
SELECT operation  
    description, [817](#)–[818](#)  
    `SELECT * FROM`, [833](#)  
    `SELECT` with `WHERE`, [817](#)  
semicolons (`;`)  
    abstract methods, [8](#), [31](#)  
    anonymous inner classes, [462](#)–[463](#)  
    enums, [49](#)  
separation of concerns principle, [745](#)  
`sequential()` method, [770](#)  
Serializable interface, [302](#), [309](#)  
serialization, [301](#)  
    collections, [313](#)  
    inheritance, [309](#)–[312](#)  
    key points, [317](#)  
    object graphs, [303](#)–[306](#)  
    `ObjectOutputStream` and `ObjectInputStream`, [301](#)–[303](#)  
    static variables, [313](#)  
    transient variables, [345](#)–[346](#)  
    `writeObject()` and `readObject()`, [306](#)–[309](#)  
serialization process, [252](#)  
`setAttribute()` method, [286](#)

setLastModified() method, [282](#)  
setName() method, [657](#)  
setPosixFilePermissions() method, [287](#)  
setPriority() method  
    key points, [666](#), [702](#)  
    threads, [658](#)  
setProperty() method, [230](#)  
sets and Set interface, [347](#)  
    characters in globs, [295](#)  
    description, [349](#)  
    methods, [393](#)  
    overview, [352](#)–[353](#), [378](#)–[379](#)  
setters encapsulation, [71](#)  
setTimes() method, [284](#), [287](#)  
shadowed variables, [42](#)  
short-circuiting operations, [576](#)  
short type  
    ranges, [39](#)  
    wrappers, [361](#)  
shutdown of ExecutorService, [754](#)–[755](#)  
shutdownNow() method, [755](#)  
side effects  
    assertions, [178](#)–[179](#)  
    lambda expressions, [506](#)–[507](#)  
    parallel streams, [773](#)–[774](#)  
    streams, [588](#), [593](#)  
signalAll() method, [732](#)  
signatures of methods, [100](#), [106](#)  
signed numbers, [38](#)  
SimpleFileVisitor class, [289](#)–[290](#)  
single thread pools, [752](#)  
singleton design pattern, [128](#)  
    benefits, [133](#)  
    description, [129](#)  
    key points, [153](#)  
    problem, [129](#)–[130](#)

solution, 130–133

size  
arrays, 45  
numbers, 39

size() method  
collections, 349, 737  
lists, 378, 393  
maps, 393  
sets, 393

skip() method, 611, 777, 780

SKIP\_SIBLINGS result type, 292

SKIP\_SUBTREE result type, 292

slashes (/) for globs, 294–295

sleep() method  
key points, 702  
locks, 679  
overview, 661–664  
threads, 646, 658, 668, 699  
working with, 664

sleeping thread state, 664  
description, 660  
overview, 661–664

sort() method  
arrays, 372–373, 392  
collections, 363–365, 370–372, 392

SORTED characteristic for streams, 778

sorted collections, 349–351

sorted() method, 580–582, 587, 777, 780

SortedMap interface, 347

SortedSet interface, 347

sorting  
arrays, 372–373, 392  
collections, 363–372, 392  
Comparable interface, 365–367  
Comparator interface, 367–368  
key points, 431, 621–622

streams, 580–588  
source operations for pipelines, 553  
sources of streams, 588–589  
spaces in natural ordering, 394  
special-purpose queues, 739–740  
special relationships in inner classes, 452, 468  
split() method, 604  
spliterator() method, 779  
spontaneous wakeup, 698  
spreadsheets. *See* databases  
SQL (Structured Query Language), 815–816  
    closing resources, 875–879  
    injection attacks, 838  
    queries, 817–818  
    types, 843  
SQLException class  
    driver classes, 830  
    ResultSets, 859, 867–868  
SQLWarning class, 874–875  
square brackets ([] for array elements, 44  
stack  
    local variables, 41  
    threads, 747  
StandardWatchEventsKinds class, 298–299  
start() method  
    alternative, 746  
    threads, 646, 650, 654–656, 699  
starting threads, 644–646, 651–659  
starvation of threads, 686, 704  
stateful operations in parallel streams, 773–778  
stateless operations in parallel streams, 773  
Statement interface, 823–824, 835  
statements  
    constructing and using, 836–839  
    databases, 816  
    description, 835

states

- key points, 701–703
- threads, 659–661
- static constants, 12–13
- static fields, synchronizing, 680–681
- static initialization blocks, 138
- static nested classes, 450
  - key points, 477–478
  - overview, 468–469
- static variables and methods, 47
  - constructors, 119
  - inheritance, 75
  - inner classes, 458, 460–461
  - interfaces, 14–15
  - key points, 60, 154
  - locks, 679
  - overriding, 91, 146
  - overview, 139–142
  - references, 520
  - serialization, 313
  - singleton design pattern, 131
  - synchronizing, 677
- stop() method, 660
- store() method, 230
- stored procedures
  - information about, 871–872
  - invoking, 837
- stream classes, 252
- Stream interface, 770
  - methods, 570–571, 601
  - optionals, 572
- stream() method, 543, 547–548, 604
- streams
  - accumulators, 567–570
  - from arrays, 548
  - benefits, 552–553

from collections, [546](#)–[547](#)  
collectors, [600](#)–[603](#)  
counting, [599](#)  
creation methods, [551](#)–[552](#)  
element existence, [577](#)–[578](#)  
from files, [549](#)–[551](#)  
generating, [606](#)–[611](#)  
grouping and partitioning, [593](#)–[597](#)  
joining, [599](#)–[600](#)  
key points, [617](#)–[619](#)  
lazy, [555](#)–[556](#)  
map-filter-reduce operations, [560](#)–[564](#), [570](#)–[572](#)  
`maxBy()`, [600](#)  
`minBy()`, [600](#)  
`of()`, [548](#)  
operations, [556](#)–[559](#)  
optionals, [572](#)–[576](#)  
overview, [542](#)–[546](#)  
parallel. *See* parallel streams  
pipelines, [553](#)–[554](#)  
primitives, [551](#)  
reduce operation, [565](#)–[568](#)  
searching with, [576](#)–[580](#), [586](#)–[588](#)  
sorting, [580](#)–[588](#)  
sources, [588](#)–[589](#)  
of streams, [603](#)–[606](#), [623](#)  
summing and averaging, [597](#)–[599](#)  
values from, [589](#)–[593](#)

`strictfp` modifiers

- classes, [5](#)–[6](#)
- inner classes, [458](#)

`StringBuffer` class, [681](#)

`StringBuilder` class, [681](#)

strings

- converting to URIs, [269](#)
- immutable, [134](#)

Structured Query Language (SQL), 815–816

    closing resources, 875–879

    injection attacks, 838

    queries, 817–818

    types, 843

subclasses

    anonymous inner classes, 464–465

    concrete, 32–33

subdirectories, 289–293

subdividing tasks, 756–757

subMap() method, 385–386

submitting queries, 833–839

subpath() method, 274

subSet() method, 386

subtypes for reference variables, 83

sum() method, 565, 571

summingInt() method, 597, 603

super() calls for constructors, 126

super constructor arguments, 120

superclasses

    constructors, 118

    overridden methods, 91–92

    Serializable, 309

suppliers

    functional interfaces, 498–503

    methods and description, 523

supportsANSI92EntryLevelSQL() method, 869, 873

suppressed exceptions, 190–192

suspend() method, 660

symmetry of equals(), 339

synchronization

    blocks, 675–678

    code, 668–673

    key points, 703

    locks, 674–677

    methods, 35, 132, 674–675

need for, 679–681  
static methods, 677  
synchronizedList() method, 682–684, 734  
SynchronousQueue class, 738–740, 795  
System class, 226  
System.exit() method, 755  
system resources for threads, 747

## T

tables. *See* databases  
tailMap() method, 386  
tails of queues, 391  
tailSet() method, 386  
take() method  
    BlockingQueue, 739  
    LinkedTransferQueue, 740  
    WatchService, 299  
tasks  
    CPU-intensive vs. I/O-intensive, 748  
    decoupling from threads, 749–751  
    subdividing, 756–757  
TemporalAdjusters class, 207, 214  
terminal operations  
    pipelines, 553  
    streams, 553  
TERMINATE result type, 292  
test database overview, 819–822  
test() method, 497–498, 508–511  
thenComparing() method, 582–583  
this() calls for constructors, 118, 126  
this keyword, 43, 456–457  
Thread class  
    description, 644  
    extending, 647–648

methods, [646](#), [658](#), [699](#)  
thread methods, [699](#)

Thread.currentThread().isInterrupted() method, [755](#)  
Thread.interrupt() method, [755](#)  
thread-safe classes, [681](#)–[684](#)  
thread-safe code, [132](#)  
thread-safe collections, [736](#)  
ThreadLocalRandom class, [754](#)  
ThreadPoolExecutor, [752](#)  
ThreadPools, [745](#)–[746](#)  
    cached, [751](#)  
    fixed, [751](#)  
    scheduled, [752](#)  
    single, [752](#)

threads

- blocked code, [678](#)–[679](#)
- concurrency. *See* concurrency
- creating and putting to sleep, [664](#)
- deadlocks, [684](#)–[685](#)
- decoupling from tasks, [749](#)–[751](#)
- defining, [647](#)–[648](#)
- exercise, [678](#)
- instantiating, [644](#)–[646](#), [649](#)–[651](#)
- interaction, [690](#)–[695](#)
- key points, [701](#)–[704](#)
- limits, [747](#)
- livelocks, [685](#)–[686](#)
- locks, [678](#)–[679](#)
- making, [646](#)–[647](#)
- multiple, [654](#)–[657](#)
- names, [652](#)
- notifyAll(), [695](#)–[699](#)
- overview, [644](#)–[646](#)
- preventing execution, [661](#)
- priorities, [664](#)–[668](#)
- race conditions, [686](#)–[690](#)

scheduler, 658  
sleeping state, 661–664  
starting, 644–646, 651–659  
starvation, 686  
states and transitions, 659–661  
synchronization and locks, 674–677  
synchronization need, 679–681  
synchronization of code, 668–673  
thread-safe classes, 681–684  
threads of execution, 644, 648, 650  
three-dimensional arrays, 44–45  
TIME data type, 847  
time-slicing scheduler, 664  
times  
    adjustments, 213–214  
    classes, 224–226  
    examples, 219–220  
    formatting, 220–221  
    java.time.\* classes, 208–210  
    key points, 238–239  
    locales, 222–224  
    zoned, 211–213  
TIMESTAMP data type, 847  
toArray() method  
    lists, 375–376, 393  
    sets, 378, 393  
toCollection() method, 591, 596, 602  
toConcurrentMap() method, 790  
ToDoubleBiFunction interface, 517  
ToDoubleFunction interface, 517, 561  
toInstant() method, 213, 218  
ToIntBiFunction interface, 517  
ToIntFunction interface, 517, 597  
toList() method, 590, 596, 602  
ToLongBiFunction interface, 517  
ToLongFunction interface, 517

- toMap() method
  - Collectors, 602
  - parallel streams, 790
- top-level classes, 55, 450
- toSet() method
  - Collectors, 602
  - parallel streams, 790
- toString() method
  - arrays, 392
  - null tests, 852
  - objects, 866
  - overview, 333–334
  - Path, 274
- transfer() method, 740
- transient variables
  - overview, 46–47
  - serialization, 345–346
- transitions with threads, 659–661
- transitivity of equals(), 339
- tree structures, 275, 354
- TreeMap class, 347
  - methods, 386
  - navigating, 382–384
  - overview, 354
- TreeSet class, 347
  - creating, 378
  - methods, 386
  - navigating, 382–384
  - overview, 354
- try and catch feature
  - file I/O, 254
  - finally, 179–183
  - key points, 194
  - multi-catch clauses, 179–183
- try-with-resources feature
  - autocloseable resources with, 185–189, 194

working with, 877–878  
tryLock() method, 728–730  
two-dimensional arrays, 44–45  
TYPE\_FORWARD\_ONLY cursor type, 852–853, 874  
type-safe arrays, 394–395  
TYPE\_SCROLL\_INSENSITIVE cursor type, 853–854, 863  
TYPE\_SCROLL\_SENSITIVE cursor type, 853, 874  
types  
    casting. *See* casts  
    erasure, 403  
    parameters, 396–397  
    return. *See* return type

## U

UML (Unified Modeling Language), 82, 132  
UnaryOperator interface, 517–518, 523  
unboxing problems, 405  
unchecked exceptions, 90  
Unicode characters, 39  
Unified Modeling Language (UML), 82, 132  
unique Map keys, 353  
unordered collections, 349  
unordered() method, 770, 779, 788  
unordered parallel streams, 778–780  
unpredictability of threads, 655–656, 665  
unsorted collections, 349–350  
UnsupportedOperationException class, 736  
upcasting, 103  
UPDATE operation, 818  
updateFloat() method, 862  
updateObject() method, 865–866  
updateRow() method, 862–863  
updating  
    nongeneric code to generic, 398–399

ResultSets, [861–862](#)  
SQL, [818](#)  
upper case  
    natural ordering, [394](#)  
    SQL commands, [818](#)  
URIs, converting strings to, [269](#)  
URLs, JDBC, [830–832](#)  
usernames, hard-coding, [827](#)  
UTC (Coordinated Universal Time), [211–212](#)

## V

valueOf() method, [866](#)  
values  
    key points, [622–623](#)  
    from streams, [589–593](#)  
values() method, [51](#)  
var-args  
    key points, [59](#)  
    methods, [35–36](#)  
VARCHAR data type, [847](#)  
variable argument lists, [35–36](#)  
variables  
    access. *See* access and access modifiers  
    atomic, [722–725](#)  
    declarations, [37–39, 59–60](#)  
    enums, [50–51](#)  
    final, [45](#)  
    inner classes, [459](#)  
    instance, [40–41](#)  
    lambda expressions, [494–496](#)  
    local. *See* local variables  
    primitives, [37–39](#)  
    static, [47, 139–142](#)  
    transient, [46–47](#)

Vector class, [347](#), [352](#)  
versions of JDBC drivers, [832](#)–[833](#)  
vertical bars (|) for multi-catch clauses, [180](#)  
visibility, access, [4](#), [29](#)  
visitFile() method, [290](#)–[291](#)  
visitFileFailed() method, [291](#)  
void return type, [114](#)  
volatile keyword for threads, [689](#)

## W

wait() method  
    Class, [699](#)  
    description, [333](#)  
    key points, [704](#)  
    locks, [679](#)  
    loops, [697](#)–[699](#)  
    Object, [753](#)  
    threads, [659](#), [690](#)–[695](#)  
waiting thread state, [660](#)  
walkFileTree() method, [289](#)–[290](#)  
warnings  
    vs. fails, [402](#)  
    JDBC, [873](#)–[879](#)  
WatchKeys, [299](#)  
WatchService, [297](#)–[300](#)  
weakly consistent iterators, [737](#)  
wildcards  
    generics, [426](#)  
    globs, [294](#)–[296](#)  
    LIKE operator, [841](#)  
with() method, [221](#)  
work stealing, [759](#)–[760](#)  
wrapped classes  
    I/O, [258](#)

overview, [251–252](#)  
primitives, [358–362](#)  
write() method, [251, 256, 258, 748](#)  
write permissions, [281–282, 286](#)  
writeObject() method, [302, 306–309](#)  
Writer class, [260](#)  
writing attributes, [280–282](#)

## X

XML, [843](#)  
XOR (exclusive-OR) operator, [343](#)  
Xss1024k option, [747](#)

## Y

yield() method  
key points, [702](#)  
locks, [679](#)  
overview, [664–668](#)  
threads, [646, 658, 699](#)

## Z

zoned dates and times, [207, 211–213](#)  
ZonedDateTime class, [207, 212–214, 218, 226](#)  
ZonedDateTime class, [212–214](#)



# Beta Test Oracle Software

Get a first look at our newest products—and help perfect them. You must meet the following criteria:

- ✓ Licensed Oracle customer or Oracle PartnerNetwork member
- ✓ Oracle software expert
- ✓ Early adopter of Oracle products

Please apply at: [pdpm.oracle.com/BPO/userprofile](http://pdpm.oracle.com/BPO/userprofile)

**ORACLE®**

If your interests match upcoming activities, we'll contact you. Profiles are kept on file for 12 months.

Copyright © 2014, Oracle and/or its affiliates. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates.



# Join the Largest Tech Community in the World



Download the latest software, tools,  
and developer templates



Get exclusive access to hands-on  
trainings and workshops



Grow your professional network through  
the Oracle ACE Program



Publish your technical articles – and  
get paid to share your expertise

Join the Oracle Technology Network  
Membership is free. Visit [community.oracle.com](http://community.oracle.com)

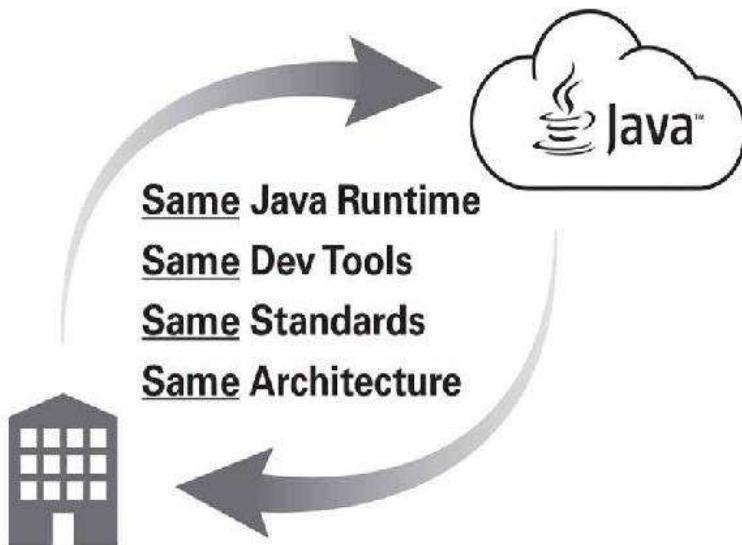
@OracleOTN

[facebook.com/OracleTechnologyNetwork](https://facebook.com/OracleTechnologyNetwork)

**ORACLE®**



# Push a Button Move Your Java Apps to the Oracle Cloud



**... or Back to Your Data Center**

**ORACLE®**

[cloud.oracle.com/java](http://cloud.oracle.com/java)



# **Oracle Learning Library**

## **Created by Oracle Experts**

## **FREE for Oracle Users**

- ✓ Vast array of learning aids
- ✓ Intuitive & powerful search
- ✓ Share content, events & saved searches
- ✓ Personalize your learning dashboard
- ✓ Find & register for training events

**ORACLE®**

[oracle.com/oll](http://oracle.com/oll)





## Reach More than 640,000 Oracle Customers with Oracle Publishing Group

Connect with the Audience that Matters Most to Your Business

**ORACLE**  
MAGAZINE

**Oracle Magazine**  
The Largest IT Publication in the World  
Circulation: 325,000  
Audience: IT Managers, DBAs, Programmers, and Developers

**PROFIT**

**Profit**  
Business Insight for Enterprise-Class Business Leaders to Help Them Build  
a Better Business Using Oracle Technology  
Circulation: 90,000  
Audience: Top Executives and Line of Business Managers



**Java Magazine**  
The Essential Source on Java Technology, the Java Programming Language,  
and Java-Based Applications  
Circulation: 225,00 and Growing Steady  
Audience: Corporate and Independent Java Developers, Programmers,  
and Architects



For more information  
or to sign up for a FREE  
subscription: Scan the  
QR code to visit Oracle  
Publishing online.

Copyright © 2016, Oracle and/or its affiliates. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

**ORACLE®**

# **Single User License Terms and Conditions**

Online access to the digital content included with this book is governed by the McGraw-Hill Education License Agreement outlined next. By using this digital content you agree to the terms of that license.

**Access** To register and activate your Total Seminars Training Hub account, simply follow these easy steps.

1. Go to [\*\*hub.totalsem.com/mheclaim\*\*](http://hub.totalsem.com/mheclaim).
2. To Register and create a new Training Hub account, enter your email address, name, and password. No further information (such as credit card number) is required to create an account.
3. If you already have a Total Seminars Training Hub account, select “Log in” and enter your email and password.
4. Enter your Product Key: **m2gw-4nj6-k3zd**
5. Click to accept the user license terms.
6. Click “Register and Claim” to create your account. You will be taken to the Training Hub and have access to the content for this book.

**Duration of License** Access to your online content through the Total Seminars Training Hub will expire one year from the date the publisher declares the book out of print.

Your purchase of this McGraw-Hill Education product, including its access code, through a retail store is subject to the refund policy of that store.

The Content is a copyrighted work of McGraw-Hill Education and McGraw-Hill Education reserves all rights in and to the Content. The Work is © 2018 by McGraw-Hill Education, LLC.

**Restrictions on Transfer** The user is receiving only a limited right to use the Content for user’s own internal and personal use, dependent on purchase and continued ownership of this book. The user may not reproduce, forward, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish, or sublicense the Content or in any way commingle the Content with other third-party content, without McGraw-Hill Education’s consent.

**Limited Warranty** The McGraw-Hill Education Content is provided on an “as is” basis. Neither McGraw-Hill Education nor its licensors make any guarantees or warranties of any kind, either express or implied, including, but not limited to, implied warranties of merchantability or fitness for a particular purpose or use as to any McGraw-Hill Education Content or the information therein or any warranties as to the accuracy, completeness, currentness, or results to be obtained from, accessing or using the McGraw-Hill Education content, or any material referenced in such content or any information entered into licensee’s product by users or other persons and/or any material available on or that can be accessed through the licensee’s product (including via any hyperlink or otherwise) or as to non-infringement of third-party rights. Any warranties of any kind, whether express or implied, are disclaimed. Any material or data obtained through use of the McGraw-Hill Education content is at your own discretion and risk and user understands that it will be solely responsible for any resulting damage to its computer system or loss of data.

Neither McGraw-Hill Education nor its licensors shall be liable to any subscriber or to any user or anyone else for any inaccuracy, delay, interruption in service, error or omission, regardless of cause, or for any damage resulting therefrom.

In no event will McGraw-Hill Education or its licensors be liable for any indirect, special or consequential damages, including but not limited to, lost time, lost money, lost profits or good will, whether in contract, tort, strict liability or otherwise, and whether or not such damages are foreseen or unforeseen with respect to any use of the McGraw-Hill Education content.