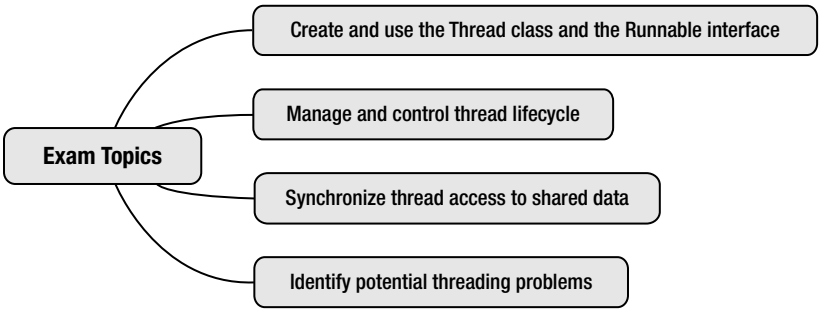
CHAPTER 13

Threads



These days, when you buy a computer—be it a laptop or a desktop—you can see labels like *dual core*, *quad core*, etc. to describe the type of processor inside the system. Processors these days have multiple cores, which are multiple execution units in the same processor. To make the best use of these multi-cores, we need to run tasks or threads in parallel. In other words, we need to make our programs multi-threaded (or concurrent). In essence, concurrency is gaining importance with more widespread use these days. Fortunately, Java has built-in support for concurrency. In this chapter, you'll learn the basics of multi-threaded programming and how to write concurrent programs and applications. More advanced topics about concurrency are covered in the next chapter.

The Latin root of the word *concurrency* means "running together." In programming, you can have multiple threads running in parallel in a program executing different tasks at the same time. Therefore, it is a powerful and useful feature.

Multiple threads can run in the context of the same process and thus share the same resources. You can use multi-threading for various reasons. In GUI applications or applets, multi-threading improves the responsiveness of the application to the users. For large computation-intensive applications, parallelizing the jobs can improve the performance of the application if it is running on multi-processor or multi-core machine.

Introduction to Concurrent Programming

In a typical application like a word processor, many tasks need to be executed at the same time—say, responding to the user, checking spellings, carrying out formatting and certain associated background tasks, etc. Executing multiple tasks at a time is expected from an interactive application like a word processor. It is possible to do such tasks sequentially; however, the user experience might not remain same. For example, many word processors have an auto-save feature. If the auto-save is invoked every 60 seconds, and if during that time the application will not respond to the user's actions, the user will feel as if the application is hanging. Instead of executing such tasks sequentially, if the auto-save task is automatically executed in the background without disrupting the main activity of responding to the user, the user experience will be much better. A similar scenario is running spell check in a dictionary in the

background as the user types some words and then suggesting alternative spelling for misspelled words. Performing such activities in parallel enhances the responsiveness of the application, and thus the user experience. Such parallel activities can be implemented as threads: running multiple threads in parallel at the same time is called *multi-threading* or *concurrency*.

Multi-threading is very useful for Internet applications as well. For example, an applet displaying stock market updates might want to retrieve the latest information and display graphs and text updates. You can write a straightforward infinite loop that will keep waiting for the updates and then refresh the graphics and text. This approach wastes processor cycles; also, the user will feel that the applet hangs when an update occurs. A better approach is to make a thread wait for the updates to occur and inform the main thread when any update happens. Then separate threads can refresh the applet graphics and text.

The `Object` and `Thread` classes and the `Runnable` interface provide the necessary support for concurrency in Java. The `Object` class has methods like `wait(), notify()/notifyAll()`, etc., which are useful for multi-threading. Since every class in Java derives from the `Object` class, all the objects have some basic multi-threading capabilities. For example, you can acquire a lock on *any* object in Java (don't worry if you don't understand yet what we mean by "acquiring a lock"—we'll discuss it later in this chapter). However, to *create* a thread, this basic support from `Object` is not useful. For that, a class should extend the `Thread` class or implement the `Runnable` interface. Both `Thread` and `Runnable` are in the `java.lang` library, so you don't have to import these classes explicitly for writing multi-threaded programs.

Important Threading-Related Methods

Table 13-1 lists some important methods in the `Thread` class, which you'll be using in this chapter.

Table 13-1. *Important Methods in the Thread Class*

Method	Method Type	Short Description
`Thread currentThread()`	Static method	Returns reference to the current thread.
`String getName()`	Instance method	Returns the name of the current thread.
`int getPriority()`	Instance method	Returns the priority value of the current thread.

`void join(),`
`void join(long),`
`void join(long, int)`
	Overloaded instance methods	The current thread invoking join on another thread waits until the other thread dies. You can optionally give the timeout in milliseconds (given in `long`) or timeout in milliseconds as well as nanoseconds (given in `long` and `int`).

`void run()`	Instance method	Once you start a thread (using the `start()` method), the `run()` method will be called when the thread is ready to execute.
`void setName(String)`	Instance method	Changes the name of the thread to the given name in the argument.
`void setPriority(int)`	Instance method	Sets the priority of the thread to the given argument value.
`void sleep(long)` `void sleep(long, int)`	Overloaded static methods	Makes the current thread sleep for given milliseconds (given in long) or for given milliseconds and nanoseconds (given in `long` and `int`).
`void start()`	Instance method	Starts the thread; JVM calls the `run()` method of the thread.
`String toString()`	Instance method	Returns the string representation of the thread; the string has the thread's name, priority, and its group.

In this chapter, you'll also be using some threading related methods in the `Object` class shown in Table 13-2.

Table 13-2. *Important Threading-Related Methods in the Object Class*

Method	Method Type	Short Description
`void wait()`, `void wait(long)`, `void wait(long, int)`	Overloaded instance methods	The current thread should have acquired a lock on this object before calling any of the wait methods. If `wait()` is called, the thread waits infinitely until some other thread notifies (by calling the `notify()`/`notifyAll()` method) for this lock. The method `wait(long)` takes milliseconds as an argument. The thread waits till it is notified or the timeout happens. The `wait(long, int)` method is similar to `wait(long)` and additionally takes nanoseconds as an argument.
`void notify()`	Instance method	The current thread should have acquired a lock on this object before calling `notify()`. The JVM chooses a *single* thread that is waiting on the lock and wakes it up.
`void notifyAll()`	Instance method	The current thread should have acquired a lock before calling `notifyAll()`. The JVM wakes up *all* the threads waiting on a lock.

Creating Threads

A Java thread can be created in two ways: by extending the `Thread` class or by implementing the `Runnable` interface. Both of them have a method named `run()`. The JVM will call this method when a thread starts executing. You can think of the `run()` method as a starting point for the execution of a thread, just like the `main()` method, which is the starting point for the execution of a program. You'll first see two examples for creating threads—extend `Thread` and implement `Runnable`—before learning the differences between them.

Extending the Thread Class

You'll first consider how to extend the `Thread` class. You need to override the `run()` method when you want to extend the `Thread` class. If you don't override the `run()` method, the default `run()` method from the `Thread` class will be called, which does nothing.

To override the `run()` method, you need to declare it as `public`; it takes no arguments and has a `void` return type—in other words, it should be declared as `public void run()`.

A thread can be created by invoking the `start()` method on the object of the `Thread` class (or its derived class). When the JVM schedules the thread, it will move the thread to a *runnable* state and then execute the `run()` method. (We'll discuss thread states later in this chapter). When the `run()` method completes its execution and returns, the thread will terminate. Listing 13-1 is the first example of multi-threading.

Listing 13-1. MyThread1.java

```
class MyThread1 extends Thread {
        public void run() {
                try {
                       sleep(1000);
                }
                catch (InterruptedException ex) {
                        ex.printStackTrace();
                        // ignore the InterruptedException - this is perhaps the one of the
                        // very few of the exceptions in Java which is acceptable to ignore
                }
                System.out.println("In run method; thread name is: "+getName());
        }
        public static void main(String args[])  {
                Thread myThread=new MyThread1();
                myThread.start();
                System.out.println("In main method; thread name is: "+
                        Thread.currentThread().getName());
        }
}
```

This program prints the following:

```
In main method; thread name is: main
In run method; thread name is: Thread-0
```

In this example, the `MyThread1` class extends the `Thread` class. You have overridden the `run()` method in this class. This `run()` method will be called when the thread runs. In the `main()` function, you create a new thread and start it using the `start()` method. An important note: you do not invoke the `run()` method directly. Instead you start the thread using the `start()` method; the `run()` method is invoked automatically by the JVM. We'll revisit this topic later.

For printing the name of the thread, you can use the instance method `getName()`, which returns a `String`. Since `main()` is a static method, you don't have access to `this` reference. So you get the current thread name using the static method `currentThread()` in the `Thread` class (which returns a `Thread` object). Now you can call `getName` on that returned object. As you'll see later, the `main()` method is also executed as a thread! However, inside the `run()` method, you can directly call the `getName()` method: `MyThread1` extends `Thread`, so all base class members are available in `MyThread1` also.

The program prints messages from both the `main` thread and `myThread` (that you created in `main`). The name of the thread printed is *Thread-0*. You'll see the default naming conventions for threads a little later.

Figure 13-1 shows how this program executes and prints the output. Note that the `main` thread and the `myThread1` thread execute at the same time (i.e., concurrently), as shown in the diagram. If you try this program a couple of times, you'll either get the output shown above, or the order of these two statements might be reversed (depending on which thread is scheduled first for executing this statement). You'll study this non-deterministic behavior a little later in this chapter.

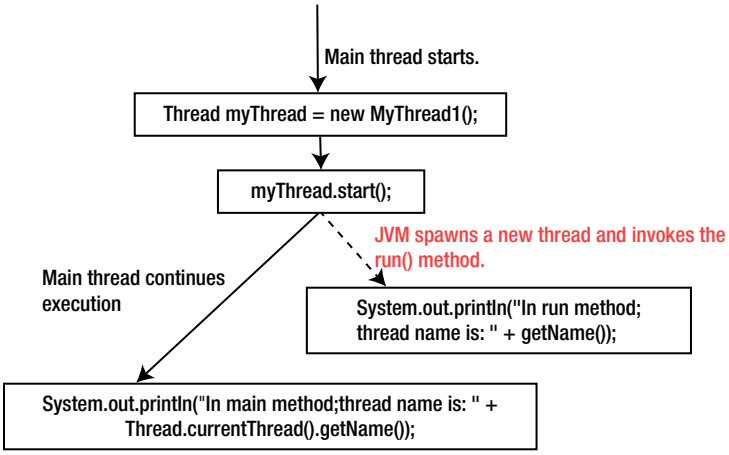


Figure 13-1. *Spawning a new thread from the main method*

Implementing the Runnable Interface

The `Thread` class itself implements the `Runnable` interface. Instead of extending the `Thread` class, you can implement the `Runnable` interface. The `Runnable` interface declares a sole method, `run()`.

```
// in java.lang package
public interface Runnable {
        public void run();
}
```

When you implement the `Runnable` interface, you need to define the `run()` method. Remember `Runnable` does not declare the `start()` method. So, how do you create a thread if you implement the `Runnable` interface? `Thread` has an overloaded constructor, which takes a `Runnable` object as an argument.

```
Thread(Runnable target)
```

You can use this overloaded constructor to create a thread from a class that implements the `Runnable` interface.

First, let's change the previous program by implementing the `Runnable` interface. If you change "`class MyThread1 extends Thread`" to "`class MyThread1 implements Runnable`" and compile the code, you get two compiler errors:

```
MyThread1.java:3: cannot find symbol
symbol  : method getName()
location: class MyThread1
                System.out.println("In run method; thread name is: " + this.getName());

MyThread1.java:7: incompatible types
found   : MyThread1
required: java.lang.Thread
                Thread myThread=new MyThread1();
```

The getName() method is available in the Thread class, but the MyThread1 class does not extend Thread any more, so it results in a compiler error. Similarly, the start() method is available in the Thread class, and you don't have that method any more since you directly implement Runnable.

Listing 13-2 contains the improved version of the program implementing the Runnable interface after fixing these two compiler errors.

Listing 13-2. MyThread2.java

```
class MyThread2 implements Runnable {
        public void run() {
                System.out.println("In run method; thread name is: "+
                Thread.currentThread().getName());
        }

        public static void main(String args[]) throws Exception {
                Thread myThread=new Thread(new MyThread2());
                myThread.start();
                System.out.println("In main method; thread name is: "+
                Thread.currentThread().getName());
        }
}
```

It prints the same output as the previous program:

```
In main method; thread name is: main
In run method; thread name is: Thread-0
```

You are implementing the run() method like the previous program. However, to get the name of the string, you must follow a round-about route and get the thread name with Thread.currentThread().getName(), as you did in the case of getting the thread name in the main() method. Similarly, in the main() method, to create a thread you must pass the object of the class to the Thread constructor. It was easy and convenient to just create the MyThread1 object and call the start() method on that while extending the Thread class.

SHOULD YOU EXTEND THE THREAD OR IMPLEMENT THE RUNNABLE?

You can either extend the Thread class or implement the Runnable interface to create a thread. So, which one do you choose?

The Thread class has the default implementation of the run() method, so if you don't provide a definition of the run() method while extending the Thread class, the compiler will not complain. However, the default implementation in the Thread class does *nothing*, so if you want your thread to do some meaningful work, you need to still define it. The Runnable interface declares the run() method, so you *must* define the run() method in your class if you implement the Runnable interface. So it doesn't matter if you implement Runnable or extend Thread. You have to define the run() method for all practical reasons. In summary, that is not a major difference between extending a Thread and implementing Runnable. How about an inheritance relationship?

Since Java supports only single inheritance, if you extend from Thread, you cannot extend from any other class. Since inheritance is an is-a relationship, you will rarely need the class to have an is-a relationship with the Thread class. So OOP purists argue that you should not extend the Thread class. On the other hand, if you

implement the `Runnable` interface, you can still extend some other class. So, many Java experts suggest that it is better to implement the `Runnable` interface unless there are some strong reasons to extend the `Thread` class.

However, extending the `Thread` class is more convenient in many cases. In the example you saw for getting the name of the thread, you had to use `Thread.currentThread().getName()` when implementing the `Runnable` interface whereas you just used the `getName()` method directly while extending `Thread` since `MyThread1 extends Thread`. So, extending `Thread` is a little more convenient in this case.

Both the techniques are useful and mostly equivalent for problem solving. So take a practical perspective here: use either of them as needed for the specific problem you are trying to solve. For the OCPJP 7 exam, you'll have to know how to create classes for threading either by extending the `Thread` class or implementing the `Runnable` interface, as well as the difference between the two approaches.

The Start() and Run() Methods

You override the `run()` method but invoke the `start()` method. Why can't you directly call the `run()` method? If you change the previous program by only changing `myThread.start()` to `myThread.run()`, what will happen? Listing 13-3 shows the program with this modification (plus changing the name of this class to `MyThread3`).

Listing 13-3. MyThread3.java

```
class MyThread3 implements Runnable {
        public void run() {
                System.out.println("In run method; thread name is: "+
                Thread.currentThread().getName());
        }

        public static void main(String args[]) throws Exception {
                Thread myThread=new Thread(new MyThread3());
                myThread.run(); // note run() instead of start() here
                System.out.println("In main method; thread name is : "+
                Thread.currentThread().getName());
        }
}
```

This prints the following:

```
In run method; thread name is: main
In main method; thread name is: main
```

Now the output is different! If you call the `run()` method directly, it simply *executes as part of the calling thread*. It does not execute as a thread: it doesn't get scheduled and get called by the JVM. That is why the `getName()` method in the `run()` method returns "main" instead of "Thread-0." When you call the `start()` method, the thread gets scheduled and the `run()` method is invoked by the JVM when it is time to execute that thread.

Never call the `run()` method directly for invoking a thread. Use the `start()` method and leave it to the JVM to implicitly invoke the `run()` method. Calling the `run()` method directly instead of calling `start()` is a mistake and is fairly common bug.

Thread Name, Priority, and Group

You need to understand three main aspects associated with each Java thread: its *name*, *priority,* and the *thread group* to which it belongs.

Every thread has a name, which you can used to identify the thread. If you do not give a name explicitly, a thread will get a default name. The priority can vary from 1, the lowest, to 10, the highest. The priority of the normal thread is by default 5, and you can change this default priority value by explicitly providing a priority value. Every thread is part of a *thread group*. It's a rarely used feature, so we won't cover it in this book. The `toString()` method of `Thread` prints these three details, so see Listing 13-4 for a simple program to get these details.

Listing 13-4. SimpleThread.java

```
class SimpleThread {
        public static void main(String []s) {
                Thread t=new Thread();
                System.out.println(t);
        }
}
```

This program prints the following:

```
Thread[Thread-0,5,main]
```

`Thread` is the name of the class. Within "[" and "]" is the name of the thread, its priority, and the thread group. You did not give any name to the thread, so the default name `Thread-0` was given (as you create more threads, threads will be given names like `Thread-1`, `Thread-2,` etc). The default priority is 5. You created the thread in `main()`, so the default thread group is "main."

Now let's try changing the name and priority of the thread using the `setName()` and `setPriority()` methods:

```
Thread t=new Thread();
t.setName("SimpleThread");
t.setPriority(9);
System.out.println(t);
```

This code segment prints the following:

```
Thread[SimpleThread,9,main]
```

The thread has the name and priority that you gave it. You can change the name of the threads as you wish and it does not change the behavior of the program. However, you need to be careful in changing thread priority since it can affect scheduling of threads. You can programmatically access the minimum, normal, and maximum priority of the threads using the static members `MIN_PRIORITY`, `NORM_PRIORITY`, and `MAX_PRIORITY` , as shown in Listing 13-5.

Listing 13-5. ThreadPriorities.java

```
class ThreadPriorities {
        public static void main(String []s) {
                System.out.println("Minimum priority of a thread: " + Thread.MIN_PRIORITY);
                System.out.println("Normal priority of a thread: " + Thread.NORM_PRIORITY);
                System.out.println("Maximum priority of a thread: " + Thread.MAX_PRIORITY);
        }
}
```

This program prints the following:

```
Minimum priority of a thread: 1
Normal priority of a thread: 5
Maximum priority of a thread: 10
```

Using the Thread.sleep() Method

Let's say you want to implement a countdown timer for a time bomb that counts from nine to zero pausing 1 second for each count. After reaching zero, it should print "Boom!!!" You can implement this functionality by creating a thread to execute the countdown. In order to pause it for each second, you can call the `Thread.sleep` method. See Listing 13-6.

Listing 13-6. TimeBomb.java

```
class TimeBomb extends Thread {
        String [] timeStr={ "Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight",
        "Nine" };

        public void run() {
                for(int i=9; i>= 0; i--) {
                        try {
                                System.out.println(timeStr[i]);
                                Thread.sleep(1000);
                        }
                        catch(InterruptedException ie) {
                                ie.printStackTrace();
                        }
                }
        }

        public static void main(String []s) {
                TimeBomb timer=new TimeBomb();
                System.out.println("Starting 10 second count down... ");
                timer.start();
                System.out.println("Boom!!!");
        }
}
```

It prints the following with 1 second pause for printing from Nine to Zero:

```
Starting 10 second count down...
Boom!!!
Nine
Eight
Seven
Six
Five
Four
Three
Two
One
Zero
```

The program didn't quite work. The message "Boom!!!" got printed even before the countdown started! Before discussing the cause of this strange behavior, let's go over the basics of the `sleep()` method.

You use the static method `sleep()` available in the `Thread` class for putting the current thread to sleep (or pause) for a certain time period. There are two overloaded static `sleep()` methods in the `Thread` class:

```
void sleep(long)
void sleep(long, int)
```

The first version of the `sleep()` method takes milliseconds as an argument. The second version, in addition to the milliseconds, takes nanoseconds as the second argument.

The `sleep()` method throws `InterruptedException`. Since `InterruptedException` is a checked exception (it extends from the `Exception` class), you need to provide a try-catch block around `sleep()` or declare the `run()` method that `throws` the exception `InterruptedException`. However, if you declare `void run() throws InterruptedException`, you won't be overriding the `run()` method since the exception specification is different (the `run()` method does not throw any checked exceptions). So, you must provide a try-catch block to handle this exception within `run()`. What should you do to handle `InterruptedException`?

First, you need to understand what `InterruptedException` means and when it gets thrown. A thread can "interrupt" another thread, say, to request it to stop working. In that case, the thread receiving the interrupt—if it is in `sleep()` or `wait()` (which we'll revisit later)—results in throwing an `InterruptedException`. The thread receiving the interrupt can ignore the interrupt and continue execution (which is not a good idea, but it is possible to do so), or it can stop the execution. You will not interrupt other threads in the multi-threaded programs we cover in this book. So let's assume that your threads will not get any interrupts, and you'll ignore the exception and ask the thread to continue working. In other words, you'll be consciously ignoring the `InterruptedException` (after calling the `printStackTrace()` method of the exception); however, in real-world programs, you may need to handle this exception if you use a thread interrupt feature.

Coming back to the program's output, the message "Boom!!!" gets printed just after printing "Starting 10 second count down. . . " and not after counting down to zero. Why did this happen?

The `main` thread starts the execution of the timer thread by calling `timer.start()`. The `main` thread execution is independent of the execution of the `timer` thread, so it executes the next statement, which is printing "Boom!!!" to the console.

But remember that you want the `main()` method to wait until the `timer` thread completes. How do you do that? For that you'll have to learn how to use the `join()` method provided in the `Thread` class.

Using Thread's Join Method

The `Thread` class has the instance method `join()` for waiting for a thread to "die." In the `TimeBomb` program, you want the `main()` thread to wait for the timer thread to complete its execution. You can use the instance method `join()` in the `Thread` class to achieve that. Here is the improved version of the `TimeBomb` program, with changes only in the `main()` method:

```
public static void main(String []s) {
        TimeBomb timer=new TimeBomb();
        System.out.println("Starting 10 second count down... ");
        timer.start();
        try {
                timer.join();
        }
        catch(InterruptedException ie) {
                ie.printStackTrace();
        }
        System.out.println("Boom!!!");
}
```

Now the program prints the output as expected:

```
Starting 10 second count down...
Nine
Eight
Seven
Six
Five
Four
Three
Two
One
Zero
Boom!!!
```

The Thread class has three overloaded versions of the join() method:

```
void join();
void join(long);
void join(long, int);
```

If the current thread invokes join() (the first overloaded version listed here) on an instance of another thread, then the current thread waits indefinitely for that other thread to die. The next two overloaded methods take a "timeout" period as an argument; the current thread will wait for the other thread to die only until the timeout period expires. The current thread will continue execution in case the other thread doesn't complete before that timeout period. The second method takes the timeout period in milliseconds (long type value) and the third overloaded version takes both milliseconds as well as nanoseconds (long and int type values).

The join() method also throws InterruptedException; you'll ignore this exception for the same reasons discussed for the sleep() method earlier in this chapter.

Asynchronous Execution

In the previous program, you saw that the main thread and the thread that you created execute independently. In other words, threads run *asynchronously*. Threads do not run sequentially (like function calls), so the order of execution of threads is not predictable—in other words, thread behavior is *non-deterministic* in nature. To understand this, consider Listing 13-7.

Listing 13-7. AsyncThread.java

```
class AsyncThread extends Thread {
        public void run() {
                System.out.println("Starting the thread " + getName());
                for(int i=0; i<3; i++) {
                        System.out.println("In thread " + getName() + "; iteration " + i);
                        try {
                                // sleep for sometime before the next iteration
                                Thread.sleep(10);
                        }
                        catch(InterruptedException ie) {
                                // we're not interrupting any threads
```

```
                                // - so safe to ignore this exeception
                                ie.printStackTrace();
                        }
                }
        }

        public static void main(String args[]) {
                AsyncThread asyncThread1=new AsyncThread();
                AsyncThread asyncThread2=new AsyncThread();
                // start both the threads around the same time
                asyncThread1.start();
                asyncThread2.start();
        }
}
```

In Listing 13-7, the `run()` method has a `for` loop that iterates three times. In the `for` loop, you print the name of the thread and the iteration number. After printing this info, you force the current thread to `sleep` for 10 milliseconds.

In one sample run, the output was the following:

```
Starting the thread Thread-0
Starting the thread Thread-1
In thread Thread-1; iteration 0
In thread Thread-0; iteration 0
In thread Thread-1; iteration 1
In thread Thread-0; iteration 1
In thread Thread-0; iteration 2
In thread Thread-1; iteration 2
```

In another sample run, the output was the following:

```
Starting the thread Thread-0
Starting the thread Thread-1
In thread Thread-1; iteration 0
In thread Thread-0; iteration 0
In thread Thread-1; iteration 1
In thread Thread-0; iteration 1
In thread Thread-1; iteration 2
In thread Thread-0; iteration 2
```

As you can see, the output for these two runs is slightly different (see the italicized part in the outputs)! Why?

The threads `Thread-0` and `Thread-1` are executed independently. The output is not fixed and the execution order of the iterations in the threads is not predictable. A programmer cannot determine the execution order of the threads. The underlying platform may use any one of the multiple processors or time-slice a single processor to allot CPU time for a thread. This cannot be controlled by the JVM or the programmer. This is one of the fundamental and most important concepts to understand in multi-threading.

You can neither predict nor control the order of execution of threads!

Since behavior of multi-threaded programs is non-deterministic, you must be careful in writing multi-threaded programs. You cannot expect pre-determined output based on the execution order of threads.

The States of a Thread

A thread has various states during its lifetime. It is important to understand the different states of a thread and learn to write robust code based on that understanding. You'll see three thread states—*new*, *runnable* and *terminated*—which are applicable to almost all threads you will create in this section. We will discuss more thread states later.

A program can access the state of the thread using `Thread.State` enumeration. The `Thread` class has the `getState()` instance method, which returns the current state of the thread; see Listing 13-8 for an example.

Listing 13-8. BasicThreadStates.java

```
class BasicThreadStates extends Thread {
        public static void main(String []s) throws Exception {
                Thread t=new Thread(new BasicThreadStates());
                System.out.println("Just after creating thread; \n" +
                                "       The thread state is: " + t.getState());
                t.start();
                System.out.println("Just after calling t.start(); \n" +
                                "       The thread state is: " + t.getState());
                t.join();
                System.out.println("Just after main calling t.join(); \n" +
                                "       The thread state is: " + t.getState());
        }
}
```

This program prints the following:

```
Just after creating thread;
        The thread state is: NEW
Just after calling t.start();
        The thread state is: RUNNABLE
Just after main calling t.join();
        The thread state is: TERMINATED
```

Just after the creation of the thread and just before calling the `start()` method on that thread, the thread is in the *new* state. After calling the `start()` method, the thread is ready to run or is in the running state (which you cannot determine), so it is in *runnable* state. From the `main()` method, you are calling `t.join()`. The `main()` method waits for the thread `t` to die. So, once the statement `t.join()` successfully gets executed by the `main()` thread, it means that the thread `t` has died or terminated. So, the thread is in the *terminated* state now.

A word of advice: be careful about accessing the thread states using the `getState()` method. Why? By the time you acquire information on a thread state and print it, the state could have changed! We know the last statement is

confusing. To understand the problem with getting thread state information using the `getState()` method, consider the previous example. In one sample run of the same program, it printed the following:

```
Just after creating thread;
        The thread state is: NEW
Just after calling t.start();
        The thread state is: TERMINATED
Just after main calling t.join();
        The thread state is: TERMINATED
```

Note the italicized part of the output, the statement after printing "`Just after calling t.start();`". In the initial output, you got the thread state (as expected) as `RUNNABLE` state. However, in another execution of the same program without any change, it printed the state as `TERMINATED`. Why? In this case, the thread is dead before you could get a chance to check it and print its status! (Note that you have not implemented the `run()` method in the `BasicThreadStates` class, so the default implementation of the `run()` method does nothing and terminates quickly.)

Every Java thread goes through these three states, as shown in Figure 13-2. Among these, the *runnable* state actually consists of two separate states at the operating system level, which we will discuss now.

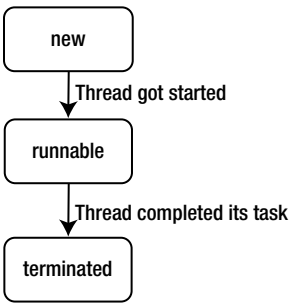


Figure 13-2. *Basic states in the life of a thread*

Two States in "Runnable" State

Once a thread makes the state transition from the *new* state to the *runnable* state, you can think of the thread having two states at the OS level: the *ready* state and *running* state. A thread is in the *ready* state when it is waiting for the OS to run it in the processor. When the OS actually runs it in the processor, it is in the *running* state. There might be many threads waiting for processor time. The current thread may end up taking lots of time and finally may give up the CPU voluntarily. In that case, the thread again goes back to the *ready* state. These two states are shown in Figure 13-3.

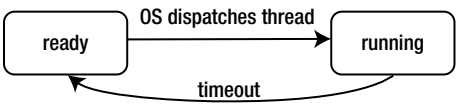


Figure 13-3. *Runnable state implemented as two states in the OS level*

Concurrent Access Problems

Concurrent programming in threads is fraught with pitfalls and problems. We will discuss two main concurrent access problems—*data races* and *deadlocks*—in this section.

Data Races

Threads share memory, and they can concurrently modify data. Since the modification can be done at the same time without safeguards, this can lead to unintuitive results.

When two or more threads are trying to access a variable and one of them wants to modify it, you get a problem known as a *data race* (also called as *race condition* or *race hazard*). Listing 13-9 shows an example of a data race.

Listing 13-9. DataRace.java

```
// This class exposes a publicly accessible counter
// to help demonstrate data race problem
class Counter {
        public static long count=0;
}

// This class implements Runnable interface
// Its run method increments the counter three times
class UseCounter implements Runnable {
        public void increment() {
                // increments the counter and prints the value
                // of the counter shared between threads
                Counter.count++;
                System.out.print(Counter.count + "  ");
        }
        public void run() {
                increment();
                increment();
                increment();
        }
}

// This class creates three threads
public class DataRace {
        public static void main(String args[]) {
                UseCounter c=new UseCounter();
                Thread t1=new Thread(c);
                Thread t2=new Thread(c);
                Thread t3=new Thread(c);
                t1.start();
                t2.start();
                t3.start();
        }
}
```

In this program, there is a `Counter` class that has a static variable `count`. In the `run()` method of the `UseCounter` class, you increment the `count` three times by calling the `increment()` method. You create three threads in the `main()` function in the `DataRace` class and start it. You expect the program to print 1 to 9 sequentially as the threads run and

increment the counters. However, when you run this program, it does print nine integer values, but the output looks like garbage! In a sample run, we got these values:

```
3  3  5  6  3  7  8  4  9
```

Note that the values will usually be different every time you run this program; when we ran it two more times, we got these outputs:

```
3  3  5  6  3  4  7  8  9

3  3  3  6  7  5  8  4  9
```

So, what is the problem?

The expression `Counter.count++` is a write operation, and the next `System.out.print` statement has a read operation for `Counter.count`. When the three threads execute, each of them has a local copy of the value `Counter.count` and when they update the `counter` with `Counter.count++`, they need not immediately reflect that value in the main memory (see Figure 13-4). In the next read operation of `Counter.count`, the local value of `Counter.count` is printed.

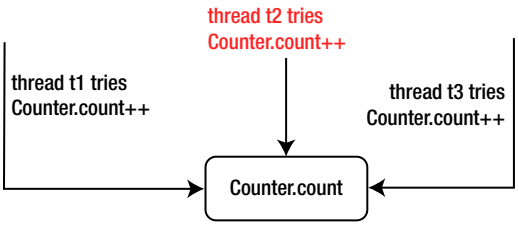


Figure 13-4. *Threads t1, t2, and t3 trying to change Counter.count, causing a data race*

Therefore, this program has a data race problem. To avoid this problem, you need to ensure that a single thread does the write and read operations together (*atomically*). The section of code that is commonly accessed and modified by more than one thread is known as *critical section*. To avoid the data race problem, you need to ensure that the critical section is executed by only one thread at a time.

How do you do that? By acquiring a lock on the object. Only a single thread can acquire a lock on an object at a time, and only that thread can execute the block of code (i.e., the critical section) protected by the lock. Until then, the other threads have to wait. Internally, this is implemented with monitors and the process is called *locking* and *unlocking* (i.e., *thread synchronization*). Let's discuss this in more detail.

Thread Synchronization

Java has a keyword, `synchronized`, that helps in thread synchronization. You can use it in two forms—synchronized blocks and synchronized methods.

Synchronized Blocks

In synchronized blocks, you use the `synchronized` keyword for a reference variable and follow it by a block of code. A thread has to acquire a lock on the synchronized variable to enter the block; when the execution of the block completes, the thread releases the lock. For example, you can acquire a lock on `this` reference if the block of code is within a non-static method:

```
synchronized(this) {
        // code segment guarded by the mutex lock
}
```

What if an exception gets thrown inside the synchronized block? Will the lock get released? Yes, no matter whether the block is executed fully or an exception is thrown, the lock will be automatically released by the JVM.

With synchronized blocks, you can acquire a lock on a reference variable only. If you use a primitive type, you will get a compiler error.

```
int i=10;
synchronized(i) { /* block of code here*/}
```

For this code, you will get the following compiler error:

```
Lock.java:5: int is not a valid type's argument for the synchronized statement
found   : int
required: reference
                synchronized(i) { /* block of code here*/}
```

Here is an improved version of the program discussed in the previous section that performs synchronized access to `Counter.count` and does both read and write operations on that in a critical section. For that, you need to change only the `increment` method, as in

```
public void increment() {
        // These two statements perform read and write operations
        // on a variable that is commonly accessed by multiple threads.
        // So, acquire a lock before processing this "critical section"
        synchronized(this) {
                Counter.count++;
                System.out.print(Counter.count + "  ");
        }
}
```

Now the program prints the expected output correctly:

```
1  2  3  4  5  6  7  8  9
```

In the `increment()` method, you acquire a lock on the `this` reference before reading and writing to `Counter.count`. So, it is not possible for more than one thread to execute these statements at the same time. Since only one thread can acquire a lock and execute the "critical section" code block, the counter is incremented by only one thread at a given time; as a result, the program prints the values 1 to 9 correctly (without the data race problem).

Synchronized Methods

An entire method can be declared synchronized. In that case, when the method declared as synchronized is called, a lock is obtained on the object on which the method is called, and it is released when the method returns to the caller. Here is an example:

```
public synchronized void assign(int i) {
        val=i;
}
```

Now the `assign()` method is a synchronized method. If you call the `assign()` method, it will acquire the lock on the `this` reference implicitly and then execute the statement `val=i;`. What happens if some other thread acquired the lock already? Just like synchronized blocks, if the thread cannot get the lock, it will be *blocked* and the thread will wait until the lock becomes available.

A synchronized method is equivalent to a synchronized block if you enclose the whole method body in a `synchronized(this)` block. So, the equivalent `assign()` method using synchronized blocks is

```
public void assign() {
        synchronized(this) {
                val=i;
        }
}
```

You can declare static methods synchronized. However, what is the reference variable on which the lock is obtained? Remember that static methods do not have the implicit `this` reference. Static synchronized methods acquire locks on the `class` object. Every class is associated with an object of `Class` type, and you can access it using `ClassName.class` syntax. For example,

```
class SomeClass {
        private static int val;
        public static synchronized void assign(int i) {
                val=i;
        }
        // more members ...
}
```

In this case, the `assign` method acquires a lock on the `SomeClass.class` object when it is called. Now the equivalent `assign()` method using synchronized blocks can be written as

```
class SomeClass {
        private static int val;
        public static void assign(int i) {
                synchronized(SomeClass.class) {
                        val=i;
                }
        }
        // more members ...
}
```

You cannot declare constructors synchronized; it will result in a compiler error. For example, for

```
class Synchronize {
        public synchronized Synchronize() { /* constructor body */}
        // more methods
}
```

you get this error:

```
Synchronize.java:2: modifier synchronized not allowed here
        public synchronized Synchronize() { /* constructor body */}
```

Why can't you declare constructors synchronized? The JVM ensures that only one thread can invoke a constructor call (for a specific constructor) at a given point in time. So, there is no need to declare a constructor synchronized. However, if you want, you can use synchronized blocks inside constructors.

Let's get back to the `Counter` example. The `increment()` method can be rewritten as a synchronized method also:

```
// declaring the increment synchronized instead of using
// a synchronized statement for a block of code inside the method
public synchronized void increment() {
        Counter.count++;
        System.out.print(Counter.count + "  ");
}
```

Now the program prints the expected output correctly:

```
1  2  3  4  5  6  7  8  9
```

In this case, `increment()` is an instance method. What about static methods? First, let's look at the data race problem when the `increment()` method is a static method; see Listing 13-10.

Listing 13-10. DataRace.java

```
class Counter {
        public static long count=0;
}

class UseCounter implements Runnable {
        public static void increment() {
                Counter.count++;
                System.out.print(Counter.count + "  ");
        }
        public void run() {
                increment();
                increment();
                increment();
        }
}
```

```
public class DataRace {
        public static void main(String args[]) {
                UseCounter c=new UseCounter();
                Thread t1=new Thread(c);
                Thread t2=new Thread(c);
                Thread t3=new Thread(c);
                t1.start();
                t2.start();
                t3.start();
        }
}
```

Yes, this program has the data race problem. To fix it, you can declare the static `increment` method as synchronized, as in

```
public static synchronized void increment() {
        Counter.count++;
        System.out.print(Counter.count + "  ");
}
```

With this change, the program does not have the data race problem.

Beginners commonly misunderstand that a synchronized block obtains a lock for a block of code. Actually, the lock is obtained for an object and not for a piece of code. The obtained lock is held until all the statements in that block complete execution.

Synchronized Blocks vs. Synchronized Methods

As you can see from the previous discussion on synchronized blocks and synchronized methods, you can use either of them to solve the data race problem. So which one should you choose? As in other language features, you need to choose between synchronized methods and blocks depending on the needs of a particular situation. Here are some factors for consideration.

If you want to acquire a lock on an object for only a small block of code and not the whole method, then synchronized blocks are sufficient; using synchronized methods is overkill in that case. In general, it is better to acquire locks for small segments of code instead of locking methods unnecessarily, so synchronized blocks are useful there. In synchronized blocks, you can explicitly provide the reference object on which you want to acquire a lock. However, in the case of a synchronized method, you do not provide any explicit reference to acquire a lock on. A synchronized method acquires an implicit lock on the `this` reference (for instance methods) and class object (for static methods).

On the other hand, if you want to acquire a lock on the entire body of a small method, then using synchronized as a method attribute is more elegant and convenient than synchronized blocks. In synchronized methods, while reading the declaration of the method itself, it becomes clear that a method is synchronized; with synchronized blocks, you need to read the documentation or look inside the code to understand that some synchronization is performed.

Deadlocks

Obtaining and using locks is tricky, and it can lead to lots of problems. One of the difficult (and common) problems is known as a *deadlock*. There are other problems such as *livelocks* and *lock starvation*, which we'll briefly discuss in the next section.

A deadlock arises when locking threads result in a situation where they cannot proceed and thus wait indefinitely for others to terminate. Say, one thread acquires a lock on resource r1 and waits to acquire another on resource r2. At the same time, say there is another thread that has already acquired r2 and is waiting to obtain a lock on r1. Neither of the threads can proceed until the other one releases the lock, which never happens—so they are stuck in a deadlock.

Listing 13-11 shows how this situation can arise (using the example from the Cricket game).

Listing 13-11. DeadLock.java

```
// Balls class has a globally accessible data member to hold the number of balls thrown so far
class Balls {
        public static long balls=0;
}

// Runs class has a globally accessible data member to hold the number of runs scored so far
class Runs {
        public static long runs=0;
}

// Counter is a thread class that has two methods - IncrementBallAfterRun and
// IncrementRunAfterBall.
// For demonstrating deadlock, we call these two methods in the run method,
// so that locking can be requested in opposite order in these two methods
class Counter implements Runnable {
        // this method increments runs variable first and then increments the balls variable
        // since these variables are accessible from other threads,
        // we need to acquire a lock before processing them
        public void IncrementBallAfterRun() {
                // since we're updating runs variable first, lock the Runs.class reference first
                synchronized(Runs.class) {
                        // now acquire lock on Balls.class variable before updating balls variable
                        synchronized(Balls.class) {
                                Runs.runs++;
                                Balls.balls++;
                        }
                }
        }

        public void IncrementRunAfterBall() {
                // since we're updating balls variable first, lock the Balls.class reference first
                synchronized(Balls.class) {
                        // now acquire lock on Runs.class variable before updating runs variable
                        synchronized(Runs.class) {
                                Balls.balls++;
                                Runs.runs++;
                        }
                }
        }
```

```
        public void run() {
                // call these two methods which acquire locks in different order
                // depending on thread scheduling and the order of lock acquision,
                // a deadlock may or may not arise
                IncrementBallAfterRun();
                IncrementRunAfterBall();
        }
}

public class DeadLock {
        public static void main(String args[]) throws InterruptedException {
                Counter c=new Counter();
                // create two threads and start them at the same time
                Thread t1=new Thread(c);
                Thread t2=new Thread(c);
                t1.start();
                t2.start();
                System.out.println("Waiting for threads to complete execution...");
                t2.join();
                t2.join();
                System.out.println("Done.");
        }
}
```

If you execute this program, the program might run fine, or it might deadlock and never terminate (the occurrence of deadlock in this program depends on how threads are scheduled).

```
D:\>java DeadLock
Waiting for threads to complete execution...
Done.

D:\>java DeadLock
Waiting for threads to complete execution...
Done.

D:\>java DeadLock
Waiting for threads to complete execution...
[deadlock - user pressed ctrl+c to terminate the program]

D:\>java DeadLock
Waiting for threads to complete execution...
Done.
```

In this example, there are two classes, `Balls` and `Runs`, with static members called `balls` and `runs`. The `Counter` class has two methods, `IncrementBallAfterRun()` and `IncrementRunAfterBall()`. They acquire locks on the `Balls.class` and `Runs.class` in the opposite order. The `run()` method calls these two methods consecutively. The `main()` method in the `Dead` class creates two threads and starts them.

When the threads `t1` and `t2` execute, they invoke the methods `IncrementBallAfterRun` and `IncrementRunAfterBall`. In these methods, locks are obtained in opposite order. It might happen that `t1` acquires a lock on `Runs.class` and then waits to acquire a lock on `Balls.class`. Meanwhile, `t2` might have acquired the `Balls.class` and now will be waiting to acquire a lock on the `Runs.class`. Therefore, this program can lead to a deadlock (Figure 13-5).

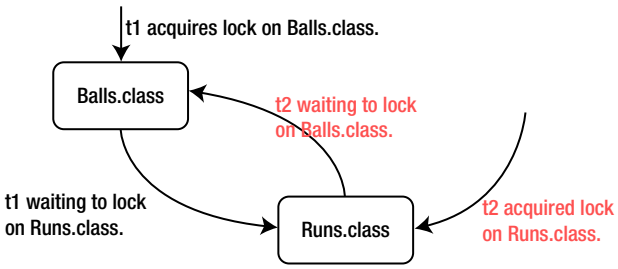


Figure 13-5. *Deadlock between threads t1 and t2*

It cannot be assured that this program will lead to a deadlock every time you execute this program. Why? You never know the sequence in which threads execute and the order in which locks are acquired and released. For this reason, such problems are said to be non-deterministic, and such problems cannot be reproduced consistently.

There are different strategies to deal with deadlocks, such as deadlock prevention, avoidance, or detection. For exam purposes, this is what you need to know about deadlocks:

- Deadlocks can arise in the context of multiple locks.
- If multiple locks are acquired in the same order, then a deadlock will not occur; however, if you acquire them in a different order, then deadlocks may occur.
- Deadlocks (just like other multi-threading problems) are non-deterministic; you cannot consistently reproduce deadlocks.

Avoid acquiring multiple locks. If you want to acquire multiple locks, make sure that they are acquired in the same order everywhere to avoid deadlocks.

Other Threading Problems

So far we discussed data races and deadlocks with examples. We'll now briefly discuss two more threading problems: livelocks and lock starvation.

Livelocks

To help understand livelocks, let's consider an analogy. Assume that there are two robotic cars that are programmed to automatically drive in the road. There is a situation where two robotic cars reach the two opposite ends of a narrow bridge. The bridge is so narrow that only one car can pass through at a time. The robotic cars are programmed such that they wait for the other car to pass through first. When both the cars attempt to enter the bridge at the same time, the following situation could happen: each car starts to enter the bridge, notices that the other car is attempting to do the same, and reverses! Note that the cars keep moving forward and backward and thus appear as if they're doing lots of work, but there is no progress made by either of the cars. This situation is called a *livelock*.

Consider two threads t1 and t2. Assume that thread t1 makes a change and thread t2 undoes that change. When both the threads t1 and t2 work, it will appear as though lots of work is getting done, but no progress is made. This situation is called a livelock in threads.

The similarity between livelocks and deadlocks is that the process "hangs" and the program never terminates. However, in a deadlock, the threads are stuck in the same state waiting for other thread(s) to release a shared resource; in a livelock, the threads keep executing a task, and there is continuous change in the process states, but the application as a whole does not make progress.

Lock Starvation

Consider the situation in which numerous threads have different priorities assigned to them (in the range of lowest priority, 1, to highest priority, 10, which is the range allowed for priority of threads in Java). When a mutex lock is available, the thread scheduler will give priority to the threads with high priority over low priority. If there are many high-priority threads that want to obtain the lock and also hold the lock for long time periods, when will the low-priority threads get a chance to obtain the lock? In other words, in a situation where low-priority threads "starve" for a long time trying to obtain the lock is known as *lock starvation*.

There are many techniques available for detecting or avoiding threading problems like livelocks and starvation, but they are not within the scope of OCPJP7 exam. From the exam perspective, you are expected to know the different kinds of threading problems that we've already covered in this chapter.

The Wait/Notify Mechanism

In multi-threaded programs, often there is a need for one thread to communicate to another thread. The wait/notify mechanism is useful when threads must communicate in order to provide a functionality.

Let's take the example of a coffee shop. A waiter is using a coffee machine in a coffee shop and delivering coffee to customers. The coffee machine in this coffee shop is an antique machine: it makes one cup of coffee at a time, and it takes five to ten minutes time to make a cup. The waiter does not have to be idle while waiting for the coffee machine to complete making coffee; he can go to customers in the meantime to deliver the coffee prepared earlier. This example is a little contrived, though: assume that coffee machine keeps making the coffee and waiter keeps delivering it.

The method `wait()` allows the calling thread to wait for the wait object (on which `wait()` is called). In other words, if you want to make a thread wait for another thread, you can ask it to wait for the wait object using the `wait()` method. A thread remains in the *wait* state until some another thread calls the `notify()` or `notifyAll()` method on the wait object. To understand the wait/notify mechanism, you are going to simulate this coffee shop situation in a program. You can implement the coffee machine as one thread and the waiter as another thread in two different classes. The coffee machine can notify the waiter to take the coffee, and it can wait until the waiter has taken the coffee from the tray. Similarly, the waiter can take the coffee if it is available and notify the coffee machine to make another cup.

Explaining the wait/notify mechanism with an example involves quite a bit of code. But this is an interesting example to illustrate this concept, so read on. Listing 13-12 contains the `CoffeeMachine` class.

Listing 13-12. CoffeeMachine.java

```
// The CoffeeMachine class runs as an independent thread.
// Once the machine makes a coffee, it notifies the waiter to pick it up.
// When the waiter asks the coffee machine to make a coffee again,
// it starts all over again, and this process keeps goes on ...
class CoffeeMachine extends Thread {
        static String coffeeMade=null;
        static final Object lock=new Object();
        private static int coffeeNumber=1;
        void makeCoffee() {
                synchronized(CoffeeMachine.lock) {
                        if(coffeeMade !=null) {
```

```
                                try {
                                        System.out.println("Coffee machine: Waiting for waiter
                                        notification to deliver the coffee");
                                        CoffeeMachine.lock.wait();
                                }
                                catch(InterruptedException ie) {
                                        ie.printStackTrace();
                                }
                        }
                        coffeeMade="Coffee No. "+coffeeNumber ++;
                        System.out.println("Coffee machine says: Made " + coffeeMade);
                        // once coffee is ready, notify the waiter to pick it up
                        CoffeeMachine.lock.notifyAll();
                        System.out.println("Coffee machine: Notifying waiter to pick the coffee ");
                }
        }

        public void run() {
                while(true) {
                        makeCoffee();
                        try {
                                System.out.println("Coffee machine: Making another coffee now");
                                // simulate the time taken to make a coffee by calling sleep method
                                Thread.sleep(10000);
                        }
                        catch(InterruptedException ie) {
                                // its okay to ignore this exception
                                // since we're not using thread interrupt mechanism
                                ie.printStackTrace();
                        }
                }
        }
}
```

The `CoffeeMachine` object is going to run as a thread, so it extends the `Thread` class and implements the `run()` method. The `run()` method goes on forever and keeps calling `the makeCoffee()` method. For each iteration, it calls `sleep()` for ten seconds to simulate the time taken for the coffee machine to make the coffee.

The `CoffeeMachine` has three static members. The `coffeeMade` member has the string description for the coffee that it has made. The `lock` member is for the synchronization between the `CoffeeMachine` and `Waiter` threads. The `numOfCoffees` is used internally by the `makeCoffee()` method to get the description of the coffee made.

The `makeCoffee()` method does most of the work. The first thing it does is acquire the lock `CoffeeMachine.lock` using the `synchronized` keyword. Inside the block, it checks if the `coffeeMade` is `null` or not. The first time the `CoffeeMachine` thread calls the `makeCoffee()` method, `coffeeMade` will be `null`. In other cases, it is the `Waiter` thread that makes `coffeeMade null` and notifies (using the `notifyAll()` method) the `CoffeeMachine` thread. If the `Waiter` thread hasn't cleared it yet, it goes to the `wait()` state and prints the message, "Waiting for waiter notification to deliver the coffee".

Once the `Waiter` notifies the `CoffeeMachine` thread, the machine delivers the next coffee to the waiter; it prints the message notifying the waiter to pick up the coffee. Now let's look at the `Waiter` class (see Listing 13-13).

Listing 13-13. Waiter.java

```
// The Waiter runs as an independent thread
// It interacts with the CoffeeMachine to wait for a coffee
// and deliver the coffee once ready and request the coffee machine
// for the next one, and this activity keeps going on forever ...
class Waiter extends Thread {
        public void getCoffee() {
                synchronized(CoffeeMachine.lock) {
                        if(CoffeeMachine.coffeeMade == null) {
                                try {
                                        // wait till the CoffeeMachine says (notifies) that
                                        // coffee is ready
                                        System.out.println("Waiter: Will get orders till
                                                        coffee machine notifies me ");
                                        CoffeeMachine.lock.wait();
                                }
                                catch(InterruptedException ie) {
                                        // its okay to ignore this exception
                                        // since we're not using thread interrupt mechanism
                                        ie.printStackTrace();
                                }
                        }
                        System.out.println("Waiter: Delivering " + CoffeeMachine.coffeeMade);
                        CoffeeMachine.coffeeMade=null;
                        // ask (notify) the coffee machine to prepare the next coffee
                        CoffeeMachine.lock.notifyAll();
                        System.out.println("Waiter: Notifying coffee machine to make another one");
                }
        }

        public void run() {
                // keep going till the user presses ctrl-C and terminates the program
                while(true) {
                        getCoffee();
                }
        }
}
```

The `Waiter` class also `extends` the `Thread` since a `Waiter` object is going to `run` as a thread as well. It has a `run()` method and it does something very simple: it keeps calling the `getCoffee()` method forever.

The `Waiter` class has the `getCoffee()` method where most of the work is done. The first thing the method does is try to acquire a lock on `CoffeeMachine.lock`. Once it gets the lock, it checks if the `coffeeMade` is `null`. If the variable is `null`, it means the `CoffeeMachine` thread is still preparing the coffee. In that case, the `Waiter` thread calls `wait()` and then prints the message, "Will get orders till coffee machine notifies me". When the `CoffeeMachine` thread has made the coffee, it will set the variable `coffeeMade`, and it will be non-null then; that thread will also notify the `Waiter` thread using `notifyAll()`.

Once the `Waiter` thread gets notified, it can deliver the coffee to the customer; it prints the message "Delivering coffee". After that, it clears the `coffeeMade` variable to `null` and notifies the `CoffeeMachine` to make another coffee ("Notifying coffee machine to make another one"). Listing 13-14 shows the `CoffeeShop` class.

Listing 13-14. CoffeeShop.java

```
// This class instantiates two threads - CoffeeMachine and Waiter threads
// and these two threads interact with each other through wait/notify
// till you terminate the application explicitly by pressing Ctrl-C
class CoffeeShop {
        public static void main(String []s) {
                CoffeeMachine coffeeMachine=new CoffeeMachine();
                Waiter waiter=new Waiter();
                coffeeMachine.start();
                waiter.start();
        }
}
```

What the `main()` method in the `CoffeeShop` class does is trivial: it creates `CoffeeMachine` and `Waiter` threads and starts them. Now, these two threads communicate with each other and go on forever. The program output looks like this:

```
Coffee machine says: Made Coffee No. 1
Coffee machine: Notifying waiter to pick the coffee
Coffee machine: Making another coffee now
Waiter: Delivering Coffee No. 1
Waiter: Notifying coffee machine to make another one
Coffee machine says: Made Coffee No. 2
Coffee machine: Notifying waiter to pick the coffee
Coffee machine: Making another coffee now
Waiter: Will get orders till coffee machine notifies me
Waiter: Delivering Coffee No. 2
Waiter: Notifying coffee machine to make another one
Coffee machine says: Made Coffee No. 3
Coffee machine: Notifying waiter to pick the coffee
Coffee machine: Making another coffee now
Waiter: Will get orders till coffee machine notifies me
Waiter: Delivering Coffee No. 3
Waiter: Notifying coffee machine to make another one
```

// goes on forever until you press Ctrl-C to terminate the application. . .

SHOULD YOU USE NOTIFY() OR NOTIFYALL()?

You have two methods—`notify()` and `notifyAll()`—for notifying (i.e., for waking up a waiting thread in the `Thread` class). But which one should you use?

Let's examine the subtle difference between these two calls. The `notify()` method wakes up *one thread* waiting for the lock (the first thread that called `wait()` on that lock). The `notifyAll()` method wakes up *all the threads* waiting for the lock; the JVM selects one of the threads from the list of threads waiting for the lock and wakes that thread up.

In the case of a single thread waiting for a lock, there is no significant difference between `notify()` and `notifyAll()`. However, when there is more than one thread waiting for the lock, in both `notify()` and `notifyAll()`, the exact thread woken up is under the control of the JVM and you cannot programmatically control waking up a specific thread.

At first glance, it appears that it is a good idea to just call `notify()` to wake up one thread; it might seem unnecessary to wake up all the threads. However, the problem with `notify()` is that the thread woken up might not be the suitable one to be woken up (the thread might be waiting for some other condition, or the condition is still not satisfied for that thread etc). In that case, the `notify()` might be lost and no other thread will wake up potentially leading to a type of deadlock (the notification is lost and all other threads are waiting for notification—forever!).

To avoid this problem, it is always better to call `notifyAll()` when there is more than one thread waiting for a lock (or more than one condition on which waiting is done). The `notifyAll()` method wakes up all threads, so it is not very efficient. However, this performance loss is negligible in real world applications.

Prefer `notifyAll()` to `notify()`.

Using `notify()`/`notifyAll()` will wake up only threads waiting on the lock on which it is called; it will not wake up any other threads. If by mistake you use `wait()` on one lock and `notify()`/`notifyAll()` on another lock, the waiting thread will never get notified and the program will hang (leading to one kind of deadlock situation)!

Let's Solve a Problem

Since the wait/notify mechanism is important to understand, let's take another example and try to understand it more rigorously.

> ***Problem Statement:*** *Assume that you need to implement a dice player game. This is a two player game (say the players are "Joe" and "Jane") where the players throw the dice on their turns. When one player throws the dice, another player waits. Once the player completes throwing, he/she informs the other player to play; after that, he/she starts waiting for the other player to throw the dice. You need to implement these two players as two threads working together. The game ends after each player throws 6 times (so there will be a total of 12 throws in the game).*

Since the problem statement says "implement these two players as two threads working together," your solution is a multi-threaded program with each player implemented as a thread. The problem also states that when one player throws the dice, another waits. So, you should perhaps use a wait/notify mechanism. The dice rolling should result in a random value, so you can use the `Random` class for creating random numbers from 1 to 6.

Here is a solution. First go through the whole program (Listing 13-15), and then you'll see the explanation of how it works.

Listing 13-15. DiceGame.java

```
import java.util.Random;

// the Gamers class just holds the name of players who roll the dice
class Gamers {
        // prevent instantiating this utility class by making constructor private
        private Gamers() {}
        public static final String JOE="Joe";
        public static final String JANE="Jane";
}

// the Dice class abstracts how the dice rolls and who plays it
class Dice {
        // to remember whose turn it is to roll the dice
        private static String turn=null;
        synchronized public static String getTurn() { return turn; }
        synchronized public static void setTurn(String player) { turn=player; }

        // which player starts the game
        public static void setWhoStarts(String name) { turn=name; }

        // prevent instantiating the class by making it private (we've only static members)
        private Dice() { }

        // when we roll the dice, it should give a random result
        private static Random random=new Random();
        // random.nextInt(6) gives values from 0 to 5, so add 1 to result in roll()
        public static int roll() { return random.nextInt(6)+1; }
}

// the class Player abstracts a player playing the Dice game
// each player runs as a separate thread, so Player extends Thread class
class Player extends Thread {
        private String currentPlayer=null;
        private String otherPlayer=null;

        public Player(String thisPlayer) {
                currentPlayer=thisPlayer;
                // we've only two players; we remember them in currentPlayer and otherPlayer
                otherPlayer=thisPlayer.equals(Gamers.JOE) ? Gamers.JANE: Gamers.JOE;
        }

        public void run() {
                // each player rolls the dice 6 times in the game
                for(int i=0; i<6; i++) {
                        // acquire the lock before proceeding
                        synchronized(Dice.class) {
```

```
                                // if its not currentPlayer's turn, then
                                // wait for otherPlayers's notification
                                while(!Dice.getTurn().equals(currentPlayer)) {
                                        try {
                                                Dice.class.wait(1000);
                                                System.out.println(currentPlayer +
                                                        " was waiting for "+otherPlayer);
                                        }
                                        catch(InterruptedException ie) {
                                                ie.printStackTrace();
                                        }
                                }
                                // its currentPlayer's turn now; throw the dice
                                System.out.println(Dice.getTurn()+" throws " + Dice.roll());
                                // set the turn to otherPlayer, and notify the otherPlayer
                                Dice.setTurn(otherPlayer);
                                Dice.class.notifyAll();
                        }
                }
        }
}

// class DiceGame just starts the game by starting player threads
class DiceGame {
        public static void main(String []s) {
                Player player1=new Player(Gamers.JANE);
                Player player2=new Player(Gamers.JOE);
                // don't forget to set who starts the game
                Dice.setWhoStarts(Gamers.JOE);
                player1.start();
                player2.start();
        }
}
```

When you run the program, the sample output will be like this:

```
Joe throws 2
Jane was waiting for Joe
Jane throws 5
Joe throws 6
Jane was waiting for Joe
Jane throws 1
Joe throws 2
Jane was waiting for Joe
Jane throws 6
Joe throws 6
Jane was waiting for Joe
Jane throws 5
Joe was waiting for Jane
Joe throws 5
Jane was waiting for Joe
```

```
Jane throws 4
Joe was waiting for Jane
Joe throws 4
Jane was waiting for Joe
Jane throws 5
```

Now, let's look at the code in more detail to understand how it works.

```
// the Gamers class just holds the name of players who roll the dice
class Gamers {
        // prevent instantiating this utility class by making constructor private
        private Gamers() {}
        public static final String JOE="Joe";
        public static final String JANE="Jane";
}
```

The class `Gamers` is just a utility class that holds the name of the players (`Joe` and `Jane`). Since there is no need to instantiate the class, you declare the constructor `private`.

The class `Dice` abstracts how the dice are rolled; it also remembers the turns that the players take.

```
class Dice {
        // to remember whose turn it is to roll the dice
        private static String turn=null;
        synchronized public static String getTurn() { return turn; }
        synchronized public static void setTurn(String player) { turn=player; }

        // which player starts the game
        public static void setWhoStarts(String name) { turn=name; }

        // prevent instantiating the class by making it private (we've only static members)
        private Dice() { }

        // when we roll the dice, it should give a random result
        private static Random random=new Random();
        // random.nextInt(6) gives values from 0 to 5, so add 1 to result in roll()
        public static int roll() { return random.nextInt(6)+1; }
}
```

You have a member named `turn` of type `String`. This variable holds the name of the current player whose turn has come to roll the dice. The method `getTurn()` and `setTurn()` are getter and setter methods for this member. When the game starts, you should say who should start the game (you need to set `turn` to a proper initial value); you do it by calling `setWhoStarts`. All the members in the class are static, so there is no need to instantiate the class; you enforce this by making the constructor private.

The dice rolling should result in a random value in the range 1 to 6. You can use the `Random` class in the `java.util` package to get the random number. The `Random` class has an instance method of `nextInt()` that you can use to get the range of values you want. If you pass int value 6 to `nextInt`, it returns the values from 0 to 5, so you add 1 to get the value ranging from 1 to 6.

The `Player` class is where you do most of the work. The class `Player` abstracts a player playing the `Dice` game. Each player runs as a separate thread, so `Player extends` the `Thread` class. Alternatively, you could implement `Player` by implementing the `Runnable` interface. Both are equivalent and acceptable solutions.

```
class Player extends Thread {
        private String currentPlayer=null;
        private String otherPlayer=null;
```

```
        public Player(String thisPlayer) {
                currentPlayer=thisPlayer;
                // we've only two players; we remember them in currentPlayer and otherPlayer
                otherPlayer=thisPlayer.equals(Gamers.JOE) ? Gamers.JANE: Gamers.JOE;
        }
        // other members
}
```

You create two `Player` threads for each of the players. So, you remember the values in `currentPlayer` and `otherPlayer`; you set these values in the `Player` constructor.

Here is the `Player's run()` method:

```
public void run() {
        // each player rolls the dice 6 times in the game
        for(int i=0; i<6; i++) {
                // acquire the lock before proceeding
                synchronized(Dice.class) {
                        // if its not currentPlayer's turn, then
                        // wait for otherPlayers's notification
                        while(!Dice.getTurn().equals(currentPlayer)) {
                                try {
                                        System.out.println(currentPlayer +
                                                " waiting for "+otherPlayer);
                                        Dice.class.wait(1000);
                                }
                                catch(InterruptedException ie) {
                                        ie.printStackTrace();
                                }
                        }
                        // its currentPlayer's turn now; throw the dice
                        System.out.println(Dice.getTurn() +
                        " throws "+Dice.roll());
                        // set the turn to otherPlayer, and notify the otherPlayer
                        Dice.setTurn(otherPlayer);
                        Dice.class.notifyAll();
                }
        }
}
```

The `run()` method will be called for each `Player` thread. Each player rolls the dice six times, so you have a `for` loop with six iterations. In every loop iteration, you check if it's the `currentPlayer's` turn to roll the dice. If not, you make the player thread wait till the `otherPlayer` informs the `currentPlayer` that his/her turn has come. Before going to check the `turn`, you need to acquire a lock. Any common lock is good, and you use the `Dice.class` as the lock here. Once the `currentPlayer` gets the notification, he/she calls the `Dice.roll()` method. His/her turn is over now, so he/she sets the `turn` to the other player and calls `notifyAll()` to wake up the `otherPlayer` thread. You could have used the `notify()` method, but it is equally acceptable to use the `notifyAll()` method, which is better to use.

The `DiceGame` class does something very simple. It has the `main()` method and you create the `Jane` and `Joe` player objects. You set one of them to start the game. You call the `start()` methods for these two player threads to start playing.

If you want a mechanism to wait for a particular event to occur, a wait/notify mechanism is the best choice. Sometimes programmers solve this problem by using a sleep call, and they repeatedly check the condition to see if the event has occurred. This is an ineffective solution. Further, calling sleep does not release the lock (unlike wait), so a solution using sleep is prone to deadlocks. Do not use the sleep method when a wait/notify mechanism is the appropriate solution.

More Thread States

Earlier in this chapter we discussed three basic thread states: *new*, *runnable* and *terminated* states. In addition to these states, a thread can also be in *blocked*, *waiting*, *timed_waiting* states, which we'll discuss now. Figure 13-6 shows how and when the state transitions typically happen for these six states.

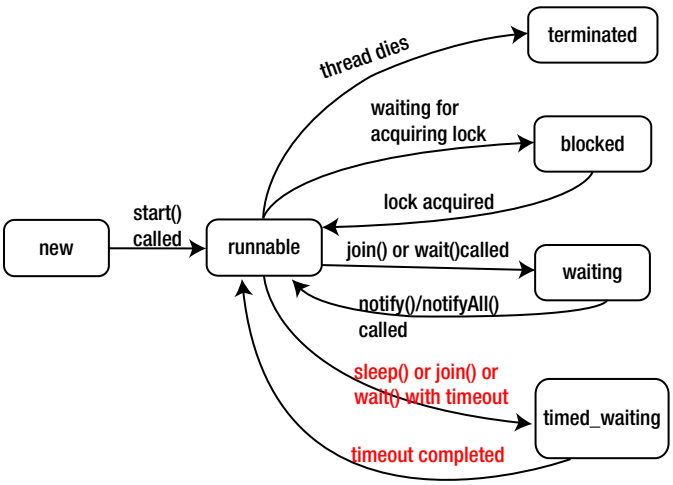


Figure 13-6. *Possible states in the lifetime of a thread*

timed_waiting and blocked States

Listing 13-16 contains a simple example to understand *timed_waiting* and *blocked* states.

Listing 13-16. MoreThreadStates.java

```
// This Thread class just invokes sleep method after acquiring lock on its class object
class SleepyThread extends Thread {
        public void run() {
                synchronized(SleepyThread.class) {
                        try {
                                Thread.sleep(1000);
                        }
                        catch(InterruptedException ie) {
                                // its okay to ignore this exception since we're not
```

```
                                // interrupting exceptions in this code
                                ie.printStackTrace();
                        }
                }
        }
}

// The class creates two threads to show how to these threads will enter into
// TIMED_WAITING and BLOCKED states
class MoreThreadStates {
        public static void main(String []s) {
                Thread t1=new SleepyThread();
                Thread t2=new SleepyThread();
                t1.start();
                t2.start();
                System.out.println(t1.getName()+": I'm in state " + t1.getState());
                System.out.println(t2.getName()+": I'm in state " + t2.getState());
        }
}
```

It prints the following:

```
Thread-0: I'm in state TIMED_WAITING
Thread-1: I'm in state BLOCKED
```

You have the `SleepyThread` class with a `run()` method that just acquires a lock and goes to sleep. You're creating two threads, `t1` and `t2`, in the `main()` method.

When `t1` runs, it acquires the lock (`SleepyThread.class`) and goes to sleep. Remember, when a thread sleeps, it doesn't relinquish the lock: it just holds the lock. So `sleep()` is called for 1 second (1000 milliseconds; the argument to `sleep()` is in milliseconds), so the thread `t1` is in state `TIMED_WAITING`.

Meanwhile, the `main` thread starts `t2` thread. When its `run()` method is called, it finds that it has to acquire the lock (`SleepyThread.class`). However, you know that the lock is already acquired by thread `t1` and the thread is still sleeping (and it is in the *timed_waiting* state). So, thread `t2` waits to acquire the lock, hence it is in the *blocking* state. The `main()` method just prints the state of these two threads by calling the `getState()` method after spawning the threads.

waiting State

The *waiting* state typically happens when a thread waits for a specific condition to happen by calling the `wait()` method. Listing 13-17 is a simple example to illustrate the *waiting* state.

Listing 13-17. WaitingThreadState.java

```
// This class has run method which waits forever since there is no other thread to notify it
class InfiniteWaitThread extends Thread {
        static boolean okayToRun=false;
        synchronized public void run() {
                while(!okayToRun) {
                        try {
                                // note the call to wait without any timeout value
                                // so it waits forever for some thread to notify it
                                wait();
                        }
```

```
                        catch(InterruptedException ie) {
                                // its okay to ignore this exception since we're not
                                // interrupting exceptions in this code
                                ie.printStackTrace();
                        }
                }
        }
}

class WaitingThreadState {
        public static void main(String []s) {
                Thread t=new InfiniteWaitThread();
                t.start();
                System.out.println(t.getName()+": I'm in state " + t.getState());
        }
}
```

This program prints the following:

```
Thread-0: I'm in state WAITING
```

You must press Ctrl+C to terminate the thread since the thread waits infinitely for the condition to happen (i.e., `okayToRun` to become `true`). In real world programs, you'll also write code to have the condition happen; in other words, you'll write code to set `okayToRun` to `true` and then call `notify()/notifyAll()`. However, since this is a dummy program just to illustrate the *waiting* state, we're leaving out that part.

What if you change the `wait` statement inside the `run` statement to, say, `wait(1000)`? Now the program will print `TIMED_WAITING`. The state *timed_waiting* happens not just for `sleep` with timeout that you saw earlier; it also works for the `wait()` method call with a timeout value.

Using Thread.State enum

The `Thread` class defines `Thread.State` enumeration, which has a list of possible thread states. Listing 13-18 is a simple program that prints the value of the states in this enumeration.

Listing 13-18. ThreadStatesEnumeration.java

```
class ThreadStatesEnumeration {
        public static void main(String []s) {
                for(Thread.State state : Thread.State.values()){
                        System.out.println(state);
                }
        }
}
```

It prints the following:

```
NEW
RUNNABLE
BLOCKED
WAITING
TIMED_WAITING
TERMINATED
```

Understanding IllegalThreadStateException

You should be cautious whenever writing code for threads, always keeping in mind the states of the threads. If you don't exercise care about the underlying states, what will happen? Let's look at the simple example in Listing 13-19 first.

Listing 13-19. ThreadStateProblem.java

```
class ThreadStateProblem {
        public static void main(String []s) {
                Thread thread=new Thread();
                thread.start();
                thread.start();
        }
}
```

The program fails with this stack trace:

```
Exception in thread "main" java.lang.IllegalThreadStateException
        at java.lang.Thread.start(Unknown Source)
        at ThreadStateProblem.main(ThreadStateProblem.java:6)
```

Here, you are trying to start a thread that has already started. When you call `start()`, the thread moves to the *new* state. There is no proper state transition from the *new* state if you call `start()` again, so the JVM throws an `IllegalThreadStateException`.

Never call the `start()` method twice on the same thread.

Can you fix the problem by adding a try-catch block around the second call to `start()`? That is a bad solution! `IllegalThreadStateException` is a `RuntimeException`, meaning that it indicates a programming error. So, you need to fix the problem in the program instead of handling it. Even if you provide a try-catch block, what can you do within the catch block? Nothing; you can leave it empty or just log the exception. Such empty catch blocks are indications of bad code. So, the correct solution in this case is to make sure that `start()` is not called again for the same thread.

Never write a catch block for handling `IllegalThreadStateException`. If you get this exception, there is certainly a bug in the code. Fix that bug.

Listing 13-20 contains another example.

Listing 13-20. ThreadStateProblem.java

```
class ThreadStateProblem extends Thread {
        public void run() {
                try {
                        wait(1000);
                }
                catch(InterruptedException ie) {
                        // its okay to ignore this exception since we're not
                        // interrupting exceptions in this code
                        ie.printStackTrace();
                }
        }

        public static void main(String []s) {
                new ThreadStateProblem().start();
        }
}
```

This program also crashes with `IllegalMonitorStateException`, like this:

```
Exception in thread "Thread-0" java.lang.IllegalMonitorStateException
        at java.lang.Object.wait(Native Method)
        at ThreadStateProblem.run(ThreadStateProblem.java:4)
```

The `wait(int)` method (with or without timeout value) should be called only after acquiring a lock: a wait() call adds the thread to the waiting queue of the acquired lock. If you don't do that, there is no proper transition from the *running* state to *timed_waiting* (or *waiting* state, if a timeout value is not given) to happen. So, the program crashes by throwing an `IllegalMonitorStateException` exception.

The correct fix is to acquire the lock before calling `wait()`. In this case, you can declare the `run()` method `synchronized`:

```
synchronized public void run() {
        try {
                wait(1000);
        }
        catch(InterruptedException ie) {
                // its okay to ignore this exception since we're not
                // interrupting exceptions in this code
                ie.printStackTrace();
        }
}
```

Since the `run()` method is `synchronized`, `wait()` will add itself to the `this` object reference lock. Since there is no one calling the `notify()`/`notifyAll()` method, after a timeout of 1 second (1000 milliseconds) is over, it will return from the `run()` method. So, the `wait(1000);` statement behaves almost like a `sleep(1000)` statement; the difference is that calling `wait()` releases the lock on this object when it waits while `sleep()` call will not release the lock when it sleeps.

Call `wait` and `notify`/`notifyAll` *only* after acquiring the relevant lock.

QUESTION TIME!

1. Here is a class named `PingPong` that extends the `Thread` class. Which of the following `PingPong` class implementations correctly prints "ping" from the worker thread and then prints "pong" from the main thread?

A.
```
class PingPong extends Thread {
        public void run() {
                System.out.println("ping ");
        }
        public static void main(String []args)  {
                Thread pingPong=new PingPong();
                System.out.print("pong");
        }
}
```

B.
```
class PingPong extends Thread {
        public void run() {
                System.out.println("ping ");
        }
        public static void main(String []args)  {
                Thread pingPong=new PingPong();
                pingPong.run();
                System.out.print("pong");
        }
}
```

C.
```
class PingPong extends Thread {
        public void run() {
                System.out.println("ping");
        }
        public static void main(String []args)  {
                Thread pingPong=new PingPong();
                pingPong.start();
                System.out.println("pong");
        }
}
```

D.
```
class PingPong extends Thread {
        public void run() {
                System.out.println("ping");
        }
        public static void main(String []args) throws InterruptedException{
                Thread pingPong=new PingPong();
                pingPong.start();
                pingPong.join();
                System.out.println("pong");
        }
}
```

Answer: D.

(The `main` thread creates the worker thread and waits for it to complete (which prints "ping"). After that it prints "pong". So, this implementation correctly prints "ping pong". Why are the other options wrong?

The `main()` method creates the worker thread, but doesn't start it. So, this program only prints "pong". The program always prints "ping pong", but it is misleading. This program directly calls the `run()` method instead of calling the `start()` method. So, this is a single threaded program. The `main` thread and the `worker` thread execute independently without any coordination. So, depending on which thread is scheduled first, you can get "ping pong" or "pong ping" printed.)

2. Consider the following program and choose the correct option describing its behavior.

```
class ThreadTest {
     public static void main(String []args) throws InterruptedException {
             Thread t1=new Thread() {
                     public void run() { System.out.print("t1 "); }
             };
             Thread t2=new Thread() {
                     public void run() { System.out.print("t2 "); }
             };
             t1.start();
             t1.sleep(5000);
             t2.start();
             t2.sleep(5000);
             System.out.println("main ");
     }
}
```

A. t1 t2 main

B. t1 main t2

C. main t2 t1

D. This program results in a compiler error.

E. This program throws a runtime error.

Answer: A. t1 t2 main

(When a new thread is created, it is in the *new* state. Then, it moves to the *runnable* state. Only from the *runnable* state can the thread go to the *timed_waiting* state after calling `sleep()`. Hence, before executing `sleep()`, the `run()` method for that thread is called. So, the program prints "t1 t2 main".)

3. You've written an application for processing tasks. In this application, you've separated the critical or urgent tasks from the ones that are not critical or urgent. You've assigned high priority to critical or urgent tasks.

   In this application, you find that the tasks that are not critical or urgent are the ones that keep waiting for an unusually long time. Since critical or urgent tasks are high priority, they run most of the time. Which one of the following multi-threading problems correctly describes this situation?

   A. Deadlock

   B. Starvation

   C. Livelock

   D. Race condition

   Answer: B. Starvation

   (The situation in which low-priority threads keep waiting for a long time to acquire the lock and execute the code in critical sections is known as starvation.)

4. Consider the following program:

```
class ExtendThread extends Thread {
         public void run() { System.out.print(Thread.currentThread().getName()); }
}

class ThreadTest{
         public static void main(String []args) throws InterruptedException {
                 Thread thread1=new Thread(new ExtendThread(), "thread1 ");
                 Thread thread2=new Thread(thread1, "thread2 ");
                 thread1.start();
                 thread2.start();
                 thread1.start();       // START
         }
}
```

   Which one of the following correctly describes the behavior of this program?

   A. The program prints the following: thread1 thread2 thread1.

   B. The program prints the following: thread1 thread1 thread1.

   C. The program prints the following: thread1 thread2.

   D. The program results in a compiler error for the statement marked with the comment `START`.

   E. The program throws an `IllegalMonitorStateException` when executing the statement marked with the comment `START`.

   Answer: E. The program throws an `IllegalMonitorStateException` when executing the statement marked with the comment `START`.

(It is illegal to call the `start()` method more than once on a thread; in that case, the thread will throw an `IllegalMonitorStateException`.]

5. Which of the following two definitions of Sync (when compiled in separate files) will compile without errors?

A.
```
class Sync {
        public synchronized void foo() {}
}
```

B.
```
abstract class Sync {
        public synchronized void foo() {}
}
```

C.
```
abstract class Sync {
        public abstract synchronized void foo();
}
```

D.
```
interface Sync {
       public synchronized void foo();
}
```

Answer: A. and B.

(Abstract methods (in abstract classes or interfaces) cannot be declared synchronized, hence the options C and D are incorrect.)

Summary

Introduction to Concurrent Programming

- You can create classes that are capable of multi-threading by implementing the `Runnable` interface or by extending the `Thread` class.
- Always implement the `run()` method. The default `run()` method in `Thread` does nothing.
- Call the `start()` method and not the `run()` method directly in code. (Leave it to the JVM to call the `run()` method.)
- Every thread has a thread name, priority, and thread-group associated with it; the default `toString()` method implementation in `Thread` prints them.
- If you call the `sleep()` method of a thread, the thread does not release the lock and it holds on to the lock.
- You can use the `join()` method to wait for another thread to terminate.
- In general, if you are not using the "interrupt" feature in threads, it is safe to ignore `InterruptedException`; however it's better still to log or print the stack trace if that exception occurs.
- Threads execute asynchronously; you cannot predict the order in which the threads run.
- Threads are also non-deterministic: in many cases, you cannot reproduce problems like deadlocks or data races every time.

Thread States

- There are three basic thread states: *new*, *runnable*, and *terminated*. When a thread is just created, it is in a *new* state; when it is ready to run or running, it is in a *runnable* state. When the thread dies, it's in *terminated* state.
- The *runnable* state has two states internally (at the OS level): *ready* and *running* states.
- A thread will be in the *blocked* state when waiting to acquire a lock. The thread will be in the *timed_waiting* state when a timeout is given for calls like wait. The thread will be in the *waiting* state when, for example, `wait()` is called (without a time out value).
- You will get an `IllegalThreadStateException` if your operations result in invalid thread state transitions.

Concurrent Access Problems

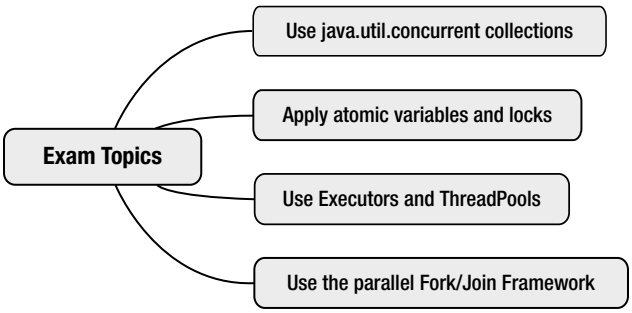
- Concurrent reads and writes to resources may lead to the *data race* problem.
- You must use thread synchronization (i.e., locks) to access shared values and avoid data races. Java provides thread synchronization features to provide protected access to shared resources—namely, synchronized blocks and synchronized methods.
- Using locks can introduce problems such as deadlocks. When a deadlock happens, the process will *hang* and will never terminate.
- A deadlock typically happens when two threads acquire locks in opposite order. When one thread has acquired one lock and waits for another lock, another thread has acquired that other lock and waits for the first lock to be released. So, no progress is made and the program deadlocks.
- To avoid deadlocks, it is better to avoid acquiring multiple locks. When you have to acquire such multiple locks, ensure that they are acquired in the same order in all places in the program.

The Wait/Notify Mechanism

- When a thread has to wait for a particular condition or event to be satisfied by another thread, you can use a wait/notify mechanism as a communication mechanism between threads.
- When a thread needs to wait for a particular condition/event, you can either call `wait()` with or without a timeout value specified.
- To avoid notifications getting lost, it is better to always use `notifyAll()` instead of `notify()`.

CHAPTER 14

Concurrency



From the beginning, Java supported concurrency in the form of low-level threads management, locks, synchronization, and APIs for concurrency. We covered them in the preceding chapter in our discussion of the `Thread` class, `Runnable` interface, and `synchronized` keyword.

Since 5.0, Java also supports high-level concurrency APIs in its `java.util.concurrent` package. In this chapter, we'll focus on these APIs for concurrent programming. These high-level APIs exploit today's multi-core hardware, in which a single processor has multiple cores. These APIs are also useful for exploiting concurrency in machines that support multiple processors.

Most of the Java concurrency utilities are provided in the `java.util.concurrent` package. Classes to efficiently update shared variables without using locks are provided in the `java.util.concurrent.atomic` subpackage. The `Lock` interface and the classes deriving from it are provided in the `java.util.concurrent.locks` subpackage.

Using java.util.concurrent Collections

There are many classes in the `java.util.concurrent` package that provide high-level APIs for concurrent programming. In this section, we will mainly discuss synchronizer classes provided in this package. Following that, we will briefly cover the important concurrent collection classes provided in the `java.util.concurrent` package.

You already understand the low-level concurrency constructs (such as the use of the `synchronized` keyword, `Runnable` interface, and `Thread` class for creating threads) from the preceding chapter. In the case of a shared resource that needs to be accessed by multiple threads, access and modifications to the shared resource need to be protected. When you use the `synchronized` keyword, you employ mutexes to synchronize between threads for safe shared access. Threads also often needed to coordinate their executions to complete a bigger higher-level task. The wait/notify pattern discussed in the last chapter is one way to coordinate the execution of multiple threads.

Using APIs for acquiring and releasing locks (using mutexes) or invoking the wait/notify methods on locks are low-level tasks. It is possible to build higher-level abstractions for thread synchronization. These high-level abstractions for synchronizing activities of two or more threads are known as *synchronizers*. Synchronizers internally make use of the existing low-level APIs for thread coordination.

The synchronizers provided in the `java.util.concurrent` library and their uses are listed here:

- `Semaphore` controls access to one or more shared resources.
- `Phaser` is used to support a synchronization barrier.
- `CountDownLatch` allows threads to wait for a countdown to complete.
- `Exchanger` supports exchanging data between two threads.
- `CyclicBarrier` enables threads to wait at a predefined execution point.

Now, we'll discuss each of these synchronizers in turn with the help of examples.

Semaphore

A semaphore controls access to shared resources. A semaphore maintains a counter to specify the number of resources that the semaphore controls. Access to the resource is allowed if the counter is greater than zero,while a zero value of the counter indicates that no resource is available at the moment and so the access is denied.

The methods `acquire()` and `release()` are for acquiring and releasing resources from a semaphore. If a thread calls `acquire()` and the counter is zero (i.e., resources are unavailable), the thread waits until the counter is non-zero and then gets the resource for use. Once the thread is done using the resource, it calls `release()` to increment the resource availability counter.

Note if the number of resources is 1, then at a given time only one thread can access the resource; in this case, using the semaphore is similar to using a lock. Table 14-1 lists the important methods in the `Semaphore` class.

Table 14-1. *Important Methods in the Semaphore Class*

Method	Description
`Semaphore(int permits)`	Constructor to create `Semaphore` objects with a given number of *permits* (the number of threads that can access the resource at a time). If the permit's value is negative, the given number of `release()` calls must happen before `acquire()` calls can succeed.
`Semaphore(int permits, boolean fair)`	Same as the previous constructor, but this extra `fair` option indicates that the permits should be allotted on a first-come-first-served basis.
`void acquire()` `void acquire(int permits)`	Acquires a permit if available; otherwise, it blocks until a permit becomes available. Can throw an `InterruptedException` if some other thread interrupts it while waiting to acquire a permit. The overloaded version takes a number of permits as an argument.
`void acquireUninterruptibly()`	Same as the `acquire()` method, but this thread cannot be interrupted while waiting to acquire a permit.
`boolean tryAcquire()` `boolean tryAcquire(long timeout, TimeUnit unit)`	Acquires a permit from the semaphore if available at the time of the call and returns `true`; if unavailable, it returns `false` immediately (without blocking). The overloaded `tryAcquire()` method additionally takes a time-out argument—the thread blocks to acquire a permit from the semaphore until a given time-out period.
`void release()` `void release(int permits)`	Releases a permit from the semaphore. The overloaded version specifies the number of permits to release.

Let's assume that there are two ATM machines available in a ATM machine room. Therefore, only two people are allowed at a time in the room. There are five people waiting outside to use the ATM machines. The situation can be simulated by the code in Listing 14-1, in which each ATM machine is treated as a resource controlled by semaphore.

Listing 14-1. ATMRoom.java

```
import java.util.concurrent.Semaphore;

// This class simulates a situation where an ATM room has only two ATM machines
// and five people are waiting to access the machine. Since only one person can access
// an ATM machine at a given time, others wait for their turn
class ATMRoom {
        public static void main(String []args) {
                // assume that only two ATM machines are available in the ATM room
                Semaphore machines = new Semaphore(2);

                // list of people waiting to access the machine
                new Person(machines, "Mickey");
                new Person(machines, "Donald");
                new Person(machines, "Tom");
                new Person(machines, "Jerry");
                new Person(machines, "Casper");
        }
}

// Each Person is an independent thread; but their access to the common resource
// (two ATM machines in the ATM machine room in this case) needs to be synchronized.
class Person extends Thread {
        private Semaphore machines;
        public Person(Semaphore machines, String name) {
                this.machines = machines;
                this.setName(name);
                this.start();
        }
        public void run() {
                try {
                        System.out.println(getName() + " waiting to access an ATM machine");
                        machines.acquire();
                        System.out.println(getName() + " is accessing an ATM machine");
                        Thread.sleep(1000); // simulate the time required for withdrawing amount
                        System.out.println(getName() + " is done using the ATM machine");
                        machines.release();
                } catch(InterruptedException ie) {
                        System.err.println(ie);
                }
        }
}
```

Here is the output of the program in one sample run:

```
Mickey waiting to access an ATM machine
Tom waiting to access an ATM machine
```

```
Jerry waiting to access an ATM machine
Donald waiting to access an ATM machine
Casper waiting to access an ATM machine
Tom is accessing an ATM machine
Mickey is accessing an ATM machine
Tom is done using the ATM machine
Mickey is done using the ATM machine
Jerry is accessing an ATM machine
Donald is accessing an ATM machine
Donald is done using the ATM machine
Jerry is done using the ATM machine
Casper is accessing an ATM machine
Casper is done using the ATM machine
```

Now let's analyze how this program works. People waiting to access an ATM machine are simulated by creating a `Person` class that extends `Thread`. The `run()` method in the `Thread` class acquires a semaphore, simulates withdrawing money from the ATM machine, and releases the semaphore.

The `main()` method simulates an ATM room with two ATM machines by creating a `Semaphore` object with two *permits*. People waiting in the queue to access the ATM machine are implemented by just adding them to the `Semaphore` object.

As you can see from the program output, the semaphore allows only two threads at a time and the other threads keep waiting. When a thread releases the semaphore, another thread acquires it. Cool, isn't it?

CountDownLatch

This synchronizer allows one or more threads to wait for a countdown to complete. This countdown could be for a set of events to happen or until a set of operations being performed in other threads completes. Table 14-2 lists important methods in this class.

Table 14-2. *Important Methods in the CountDownLatch Class*

Method	Description
`CountDownLatch(int count)`	Creates an instance of `CountDownLatch` with the number of times the `countDown()` method must be called before the threads waiting with `await()` can continue execution.
`void await()`	If the current count in `CountDownLatch` object is zero, it immediately returns; otherwise, the thread blocks until the countdown reaches zero. Can throw an `InterruptedException`.
`boolean await(long timeout, TimeUnit unit)`	Same as the previous method, `await()`, but takes an additional time-out argument. If the thread returns successfully after the count reaches zero, this method returns `true`; if the thread returns because of time-out, it returns `false`.
`void countDown()`	Reduces the number of counts by one in this `CountDownLatch` object. If the count reaches zero, all the (a)waiting threads are released. If the current count is already zero, nothing happens.
`long getCount()`	Returns the pending counts in this `CountDownLatch` object.

When you create a `CountDownLatch`, you initialize it with an integer, which represents a count value. Threads would wait (by calling the `await()` method) for this count to reach zero. Once zero is reached, all threads are released; any other calls to `await()` would return immediately since the count is already zero. The counter value can be decremented by one by calling the `countDown()` method. You can get the current value of the counter using the `getCount()` method. See Listing 14-2.

Listing 14-2. RunningRaceStarter.java

```
import java.util.concurrent.*;

// this class simulates the start of a running race by counting down from 5. It holds
// three runner threads to be ready to start in the start line of the race and once the count down
// reaches zero, all the three runners start running...

class RunningRaceStarter {
        public static void main(String []args) throws InterruptedException {
                CountDownLatch counter = new CountDownLatch(5);
                // count from 5 to 0 and then start the race

                // instantiate three runner threads
                new Runner(counter, "Carl");
                new Runner(counter, "Joe");
                new Runner(counter, "Jack");

                System.out.println("Starting the countdown ");
                long countVal = counter.getCount();
                while(countVal > 0) {
                        Thread.sleep(1000); // 1000 milliseconds = 1 second
                        System.out.println(countVal);
                        if(countVal == 1) {
                                // once counter.countDown(); in the next statement is called,
                                // Count down will reach zero; so shout "Start"
                                System.out.println("Start");
                        }
                        counter.countDown(); // count down by 1 for each second
                        countVal = counter.getCount();
                }
        }
}

// this Runner class simulates a track runner in a 100-meter dash race. The runner waits until the
// count down timer gets to zero and then starts running
class Runner extends Thread {
        private CountDownLatch timer;
        public Runner(CountDownLatch cdl, String name) {
                timer = cdl;
                this.setName(name);
                System.out.println(this.getName() + " ready and waiting for the count down to start");
                start();
        }
```

```
        public void run() {
                try {
                        // wait for the timer count down to reach 0
                        timer.await();
                } catch (InterruptedException ie) {
                        System.err.println("interrupted -- can't start running the race");
                }
                System.out.println(this.getName() + " started running");
        }
}
```

This program prints the following:

```
Carl ready and waiting for the count down to start
Joe ready and waiting for the count down to start
Jack ready and waiting for the count down to start
Starting the countdown
5
4
3
2
1
Start
Joe started running
Carl started running
Jack started running
```

Let's consider how the program works. The class `Runner` simulates a runner in a running race waiting to start running. It waits for the race to start by calling the `await()` method on the `CountDownLatch` object passed through the constructor.

The `RunningRaceStarter` class creates a `CountDownLatch` object. This counter object is initialized with the count value 5, which means the countdown is from 5 to 0. In the `main()` method, you create `Runner` objects; these three threads wait on the `counter` object. For each second, you call the `countDown()` method, which decrements count by 1. Once the count reaches zero, all three waiting threads are released and they automatically continue execution.

Note: In this program, the sequence in which Joe, Carl, or Jack is printed cannot be predicted since it depends on thread scheduling. So, if you run this program, you may get these three names printed in some other order.

Exchanger

The `Exchanger` class is meant for exchanging data between two threads. What `Exchanger` does is something very simple: it waits until both the threads have called the `exchange()` method. When both threads have called the `exchange()` method, the `Exchanger` object actually exchanges the data shared by the threads with each other. This class is useful when two threads need to synchronize between them and continuously exchange data.

This class is a tiny class with only one method: `exchange()`. Note that this `exchange()` method has an overloaded form where it takes a time-out period as an argument.

Listing 14-3 shows an example simulating silly talk between the Java Duke mascot and the coffee shop. The two threads `DukeThread` and `CoffeeShop` threads run independently. However, for a chat to happen, they need to listen when the other is talking. An `Exchange` object provides a means for them to talk to each other.

Listing 14-3. KnockKnock.java

```
import java.util.concurrent.Exchanger;

// The DukeThread class runs as an independent thread. It talks to the CoffeeShopThread that
// also runs independently. The chat is achieved by exchanging messages through a common
// Exchanger<String> object that synchronizes the chat between them.
// Note that the message printed are the "responses" received from CoffeeShopThread
class DukeThread extends Thread {
        private Exchanger<String> sillyTalk;

        public DukeThread(Exchanger<String> args) {
                sillyTalk = args;
        }
        public void run() {
                String reply = null;
                try {
                        // start the conversation with CoffeeShopThread
                        reply = sillyTalk.exchange("Knock knock!");
                        // Now, print the response received from CoffeeShopThread
                        System.out.println("CoffeeShop: " + reply);

                        // exchange another set of messages
                        reply = sillyTalk.exchange("Duke");
                        // Now, print the response received from CoffeeShopThread
                        System.out.println("CoffeeShop: " + reply);

                        // an exchange could happen only when both send and receive happens
                        // since this is the last sentence to speak, we close the chat by
                        // ignoring the "dummy" reply
                        reply = sillyTalk.exchange("The one who was born in this coffee shop!");
                        // talk over, so ignore the reply!
                } catch(InterruptedException ie) {
                        System.err.println("Got interrupted during my silly talk");
                }
        }
}

class CoffeeShopThread extends Thread {
        private Exchanger<String> sillyTalk;

        public CoffeeShopThread(Exchanger<String> args) {
                sillyTalk = args;
        }
        public void run() {
                String reply = null;
                try {
                        // exchange the first messages
                        reply = sillyTalk.exchange("Who's there?");
                        // print what Duke said
                        System.out.println("Duke: " + reply);
```

```
                        // exchange second message
                        reply = sillyTalk.exchange("Duke who?");
                        // print what Duke said
                        System.out.println("Duke: " + reply);

                        // there is no message to send, but to get a message from Duke thread,
                        // both ends should send a message; so send a "dummy" string
                        reply = sillyTalk.exchange("");
                        System.out.println("Duke: " + reply);
                } catch(InterruptedException ie) {
                        System.err.println("Got interrupted during my silly talk");
                }
        }
}

// Coordinate the silly talk between Duke and CoffeeShop by instantitaing the Exchanger object
// and the CoffeeShop and Duke threads
class KnockKnock {
                public static void main(String []args) {
                        Exchanger<String> sillyTalk = new Exchanger<String>();
                        new CoffeeShopThread(sillyTalk).start();
                        new DukeThread(sillyTalk).start();
                }
}
```

The program prints the following:

```
Duke: Knock knock!
CoffeeShop: Who's there?
Duke: Duke
CoffeeShop: Duke who?
Duke: The one who was born in this coffee shop!
```

The comments inside the program explain how the program works. The main concept to understand with this example is that `Exchanger` helps coordinate (i.e., synchronize) exchanging messages between two threads. Both the threads wait for each other and use the `exchange()` method to exchange messages.

CyclicBarrier

There are many situations in concurrent programming where threads may need to wait at a predefined execution point until all other threads reach that point. `CyclicBarrier` helps provide such a synchronization point; see Table 14-3 for the important methods in this class.

Table 14-3. *Important Methods in the CyclicBarrier Class*

Method	Description
`CyclicBarrier(int numThreads)`	Creates a `CyclicBarrier` object with the number of threads waiting on it specified. Throws `IllegalArgumentException` if `numThreads` is negative or zero.
`CyclicBarrier(int parties, Runnable barrierAction)`	Same as the previous constructor; this constructor additionally takes the thread to call when the barrier is reached.
`int await()` `int await(long timeout, TimeUnit unit)`	Blocks until the specified number of threads have called `await()` on this barrier. The method returns the arrival index of this thread. This method can throw an `InterruptedException` if the thread is interrupted while waiting for other threads or a `BrokenBarrierException` if the barrier was broken for some reason (for example, another thread was timed-out or interrupted). The overloaded method takes a time-out period as an additional option; this overloaded version throws a `TimeoutException` if all other threads aren't reached within the time-out period.
`boolean isBroken()`	Returns true if the barrier is broken. A barrier is broken if at least one thread in that barrier was interrupted or timed-out, or if a barrier action failed throwing an exception.
`void reset()`	Resets the barrier to the initial state. If there are any threads waiting on that barrier, they will throw the `BrokenBarrier` exception.

Listing 14-4 is an example that makes use of `CyclicBarrier` class.

Listing 14-4. CyclicBarrierTest.java

```
import java.util.concurrent.*;

// The run() method in this thread should be called only when four players are ready to start the game
class MixedDoubleTennisGame extends Thread {
        public void run() {
                System.out.println("All four players ready, game starts \n Love all...");
        }
}

// This thread simulates arrival of a player.
// Once a player arrives, he/she should wait for other players to arrive
class Player extends Thread {
        CyclicBarrier waitPoint;
        public Player(CyclicBarrier barrier, String name) {
                this.setName(name);
                waitPoint = barrier;
                this.start();
                }
        public void run() {
                System.out.println("Player " + getName() + " is ready ");
```

```
                try {
                        waitPoint.await(); // await for all four players to arrive
                } catch(BrokenBarrierException | InterruptedException exception) {
                         System.out.println("An exception occurred while waiting... " + exception);
                }
        }
}

// Creates a CyclicBarrier object by passing the number of threads and the thread to run
// when all the threads reach the barrier
class CyclicBarrierTest {
        public static void main(String []args) {
                // a mixed-double tennis game requires four players; so wait for four players
                // (i.e., four threads) to join to start the game
                System.out.println("Reserving tennis court \n As soon as four players arrive,
game will start");
                CyclicBarrier barrier = new CyclicBarrier(4, new MixedDoubleTennisGame());
                new Player(barrier, "G I Joe");
                new Player(barrier, "Dora");
                new Player(barrier, "Tintin");
                new Player(barrier, "Barbie");
        }
}
```

The program prints the following:

```
Reserving tennis court
As soon as four players arrive, game will start
Player G I Joe is ready
Player Dora is ready
Player Tintin is ready
Player Barbie is ready
All four players ready, game starts
 Love all...
```

Now let's see how this program works. In the `main()` method you create a `CyclicBarrier` object. The constructor takes two arguments: the number of threads to wait for, and the thread to invoke when all the threads reach the barrier. In this case, you have four players to wait for, so you create four threads, with each thread representing a player. The second argument for the `CyclicBarrier` constructor is the `MixedDoubleTennisGame` object since this thread represents the game, which will start once all four players are ready.

Inside the `run()` method for each `Player` thread, you call the `await()` method on the `CyclicBarrier` object. Once the number of awaiting threads for the `CyclicBarrier` object reaches four, the `run()` method in `MixedDoubleTennisGame` is called.

Phaser

`Phaser` is a useful feature when few independent threads have to work in phases to complete a task. So, a synchronization point is needed for the threads to work on a part of a task, wait for others to complete other part of the task, and do a sync-up before advancing to complete the next part of the task. Table 14-4 lists important methods in this class.

Table 14-4. *Important Methods in the Phaser class*

Method	Description
`Phaser()`	Creates a `Phaser` object with no registered parties and no parents. The initial phase is set to 0.
`Phaser(int numThreads)`	Creates a `Phaser` object with a given number of threads (parties) to arrive to advance to the next stage; the initial phase is set to 0.
`int register()`	Adds a new thread (party) to this `Phaser` object. Returns the phase current number. Throws an `IllegalStateException` if the maximum supported parties are already registered.
`int bulkRegister(int numThreads)`	Adds `numThreads` of unarrived parties to this `Phaser` object. Returns the phase current number. Throws an `IllegalStateException` if maximum supported parties are already registered.
`int arrive()`	Arrives at this phase without waiting for other threads to arrive. Returns the arrival phase number. Can throw an `IllegalStateException`.
`int arriveAndDeregister()`	Same as the previous method, but also deregisters from the `Phaser` object.
`int arriveAndAwaitAdvance()`	Arrive at this phase and waits (i.e., blocks) until other threads arrive.
`int awaitAdvance(int phase)`	Waits (i.e., blocks) until this `Phaser` object advances to the given phase value.
`int getRegisteredParties()`	Returns the number of threads (parties) registered with this `Phaser` object.
`int getArrivedParties()`	Returns the number of threads (parties) arrived at the current phase of the `Phaser` object.
`int getUnarrivedParties()`	Returns the number of threads (parties) that have not arrived when compared to the registered parties at the current phase of the `Phaser` object.

Consider the example of processing a delivery order in a small coffee shop. Assume that there are only three workers: a cook, a helper, and an attendant. To simplify the program logic, assume that each delivery order consists of three food items. Completing a delivery order consists of preparing the three orders one after another. To complete preparing a food item, all three workers—the cook, the helper, and the attendant—should do their part of the work. Listing 14-5 shows how this situation can be implemented using the `Phaser` class.

Listing 14-5. ProcessOrder.java

```
import java.util.concurrent.*;

// ProcessOrder thread is the master thread overlooking to make sure that the Cook, Helper,
// and Attendant are doing their part of the work to complete preparing the food items
// and complete order delivery
// To simplify the logic, we assume that each delivery order consists of exactly three food items
class ProcessOrder {
        public static void main(String []args) throws InterruptedException {
                // the Phaser is the synchronizer to make food items one-by-one,
                // and deliver it before moving to the next item
                Phaser deliveryOrder = new Phaser(1);
```

```
                System.out.println("Starting to process the delivery order ");

                new Worker(deliveryOrder, "Cook");
                new Worker(deliveryOrder, "Helper");
                new Worker(deliveryOrder, "Attendant");

                for(int i = 1; i <= 3; i++) {
                        // Prepare, mix and deliver this food item
                        deliveryOrder.arriveAndAwaitAdvance();
                        System.out.println("Deliver food item no. " + i);
                }
                // work completed for this delivery order, so deregister
                deliveryOrder.arriveAndDeregister();
                System.out.println("Delivery order completed... give it to the customer");
        }
}

// The work could be a Cook, Helper, or Attendant. Though the three work independently, the
// should all synchronize their work together to do their part and complete preparing a food item
class Worker extends Thread {
        Phaser deliveryOrder;
        Worker(Phaser order, String name) {
                deliveryOrder = order;
                this.setName(name);
                deliveryOrder.register();
                start();
        }
        public void run() {
                for(int i = 1; i <= 3; i++) {
                        System.out.println("\t" + getName() + " doing his work for order no. " + i);
                        if(i == 3) {
                                // work completed for this delivery order, so deregister
                                deliveryOrder.arriveAndDeregister();
                        } else {
                                deliveryOrder.arriveAndAwaitAdvance();
                        }
                        try {
                                Thread.sleep(3000); // simulate time for preparing the food item
                        } catch(InterruptedException ie) {
                                /* ignore exception */
                                ie.printStackTrace();
                        }
                }
        }
}
```

The program prints the following:

```
Starting to process the delivery order
        Cook doing his work for order no. 1
        Attendant doing his work for order no. 1
        Helper doing his work for order no. 1
Deliver food item no. 1
        Helper doing his work for order no. 2
        Attendant doing his work for order no. 2
        Cook doing his work for order no. 2
Deliver food item no. 2
        Helper doing his work for order no. 3
        Cook doing his work for order no. 3
        Attendant doing his work for order no. 3
Deliver food item no. 3
Delivery order completed...give it to the customer
```

In this program, you create a `Phaser` object to support the synchronizing of three `Worker` thread objects. You create a `Phaser` object by calling the default constructor of the `Phaser` object. When the `Worker` thread objects are created, they register themselves to the `Phaser` object. Alternatively, you could have called

```
Phaser deliveryOrder = new Phaser(3); // for three parties (i.e., threads)
```

In this case, you would not need to call the `register()` method on the `Phaser` object in the `Worker` thread constructor.

In this case, you've assumed that a delivery order consists of processing three food items, so the `for` loop runs three times. For each iteration, you call `deliveryOrder.arriveAndAwaitAdvance()`. For this statement to proceed, all the three parties (the `Cook`, `Helper`, and `Attendant`) have to complete their part of the work to prepare the food item. You simulate "preparing food" by calling the `sleep()` method in the `run` method for these `Worker` threads. These `worker` threads call `deliveryOrder.arriveAndAwaitAdvance()` for preparing each food item. As each food item is prepared (i.e., each phase is completed), the work progresses to the next phase. Once three phases are complete, the delivery order processing is complete and the program returns.

Concurrent Collections

The `java.util.concurrent` package provides a number of classes that are thread-safe equivalents of the ones provided in the collections framework classes in the `java.util` package (see Table 14-5). For example, `java.util.concurrent.ConcurrentHashMap` is a concurrent equivalent to `java.util.HashMap`. The main difference between these two containers is that you need to explicitly synchronize insertions and deletions with `HashMap`, whereas such synchronization is built into the `ConcurrentHashMap`. If you know how to use `HashMap`, you know how to use `ConcurrentHashMap` implicitly. From the OCPJP 7 exam perspective, you only need to have an overall understanding of the classes in Table 14-5, so we won't delve into details on how to make use of these classes.

Table 14-5. *Some Concurrent Collection Classes in the java.util.concurrent Package*

Class/Interface	Short Description
`BlockingQueue`	This interface extends the `Queue` interface. In `BlockingQueue`, if the queue is empty, it waits (i.e., blocks) for an element to be inserted, and if the queue is full, it waits for an element to be removed from the queue.
`ArrayBlockingQueue`	This class provides a fixed-sized array based implementation of the `BlockingQueue` interface.
`LinkedBlockingQueue`	This class provides a linked-list-based implementation of the `BlockingQueue` interface.
`DelayQueue`	This class implements `BlockingQueue` and consists of elements that are of type `Delayed`. An element can be retrieved from this queue only after its delay period.
`PriorityBlockingQueue`	Equivalent to `java.util.PriorityQueue`, but implements the `BlockingQueue` interface.
`SynchronousQueue`	This class implements `BlockingQueue`. In this container, each `insert()` by a thread waits (blocks) for a corresponding `remove()` by another thread and vice versa.
`LinkedBlockingDeque`	This class implements `BlockingDeque` where insert and remove operations could block; uses a linked-list for implementation.
`ConcurrentHashMap`	Analogous to `Hashtable`, but with safe concurrent access and updates.
`ConcurrentSkipListMap`	Analogous to `TreeMap`, but provides safe concurrent access and updates.
`ConcurrentSkipListSet`	Analogous to `TreeSet`, but provides safe concurrent access and updates.
`CopyOnWriteArrayList`	Similar to `ArrayList`, but provides safe concurrent access. When the `ArrayList` is updated, it creates a fresh copy of the underlying array.
`CopyOnWriteArraySet`	A `Set` implementation, but provides safe concurrent access and is implemented using `CopyOnWriteArrayList`. When the container is updated, it creates a fresh copy of the underlying array.

Listings 14-6 and 14-7 show how a concurrent version differs from its non-concurrent version. Assume that you have a `PriorityQueue` object shared by two threads. Assume that one thread inserts an element into the priority queue, and the other thread removes an element. If the threads are scheduled such that the inserting an element occurs before removing the element, there is no problem. However, if the first thread attempts to remove an element before the second thread inserts an element, you get into trouble.

Listing 14-6. PriorityQueueExample.java

```
import java.util.*;

// Simple PriorityQueue example. Here, we create two threads in which one thread inserts an element,
// and another thread removes an element from the priority queue.
class PriorityQueueExample {
        public static void main(String []args) {
                final PriorityQueue<Integer> priorityQueue = new PriorityQueue<>();
                // spawn a thread that removes an element from the priority queue
```

```
                new Thread() {
                        public void run() {
                                // Use remove() method in PriorityQueue to remove the element if available
                                System.out.println("The removed element is: " + priorityQueue.remove());
                        }
                }.start();
                // spawn a thread that inserts an element into the priority queue
                new Thread() {
                        public void run() {
                                // insert Integer value 10 as an entry into the priority queue
                                priorityQueue.add(10);
                                System.out.println("Successfully added an element to the queue ");
                        }
                }.start();
        }
}
```

If you run this program, it throws an exception like this:

```
Exception in thread "Thread-0" java.util.NoSuchElementException
at java.util.AbstractQueue.remove(AbstractQueue.java:117)
at PriorityQueueExample$1.run(QueueExample.java:10)
Successfully added an element to the queue
```

This output indicates that the first thread attempted removing an element from an empty priority queue, and hence it results in a `NoSuchElementException`.

However, consider a slight modification of this program (Listing 14-7) that uses a `PriorityBlockingQueue` instead of `PriorityQueue`.

Listing 14-7. PriorityBlockingQueueExample.java

```
// Illustrates the use of PriorityBlockingQueue. In this case, if there is no element available in
the priority queue
// the thread calling take() method will block (i.e., wait) until another thread inserts an element

import java.util.concurrent.*;

class PriorityBlockingQueueExample {
        public static void main(String []args) {
                final PriorityBlockingQueue<Integer> priorityBlockingQueue
                        = new PriorityBlockingQueue<>();
                new Thread() {
                        public void run() {
                                try {
                                        // use take() instead of remove()
                                        // note that take() blocks, whereas remove() doesn't block
                                        System.out.println("The removed element is: "
                                                + priorityBlockingQueue.take());
                                } catch(InterruptedException ie) {
                                        // its safe to ignore this exception
                                        ie.printStackTrace();
                                }
                        }
```

```
                }.start();
                new Thread() {
                        public void run() {
                                // add an element with value 10 to the priority queue
                                priorityBlockingQueue.add(10);
                                System.out.println("Successfully added an element to the queue ");
                        }
                }.start();
        }
}
```

The program prints the following:

```
Successfully added an element to the queue
The removed element is: 10
```

This program will not result in a crash as in the previous case (Listing 14-6). This is because the `take()` method will block until an element gets inserted by another thread; once inserted, the `take()` method will return that value. In other words, if you're using a `PriorityQueue` object, you need to synchronize the threads such that insertion of an element always occurs before removing an element. However, in `PriorityBlockingQueue`, the order does not matter, and no matter which operation (insertion or removal of an element) is invoked first, the program works correctly. In this way, concurrent collections provide support for safe use of collections in the context of multiple threads without the need for you to perform explicit synchronization operations.

Apply Atomic Variables and Locks

The `java.util.concurrent` package has two subpackages: `java.util.concurrent.atomic` and `java.util.concurrent.locks`. In this section we discuss these two subpackages. Unlike the rest of this chapter, which discusses high-level concurrency abstractions, both atomic variables and locks are low-level APIs. However, they provide more fine-grained control when you want to write multithreaded code.

Atomic Variables

Have you seen code that acquires and releases locks for implementing primitive/simple operations like incrementing a variable, decrementing a variable, and so on? Such acquiring and releasing of locks for such primitive operations is not efficient. In such cases, Java provides an efficient alternative in the form of atomic variables.

Here is a list of some of the classes in this package and their short description:

- `AtomicBoolean`: Atomically updatable `Boolean` value.
- `AtomicInteger`: Atomically updatable `int` value; inherits from the `Number` class.
- `AtomicIntegerArray`: An `int` array in which elements can be updated atomically.
- `AtomicLong`: Atomically updatable `long` value; inherits from `Number` class.
- `AtomicLongArray`: A `long` array in which elements can be updated atomically.
- `AtomicReference<V>`: An atomically updatable object reference of type V.
- `AtomicReferenceArray<E>`: An atomically updatable array that can hold object references of type `E` (`E` refers to be base type of elements).

Only `AtomicInteger` and `AtomicLong` extend from `Number` class but not `AtomicBoolean`. All other classes in the `java.util.concurrent.atomic` subpackage inherit directly from the `Object` class.

Of the classes in the `java.util.concurrency.atomic` subpackage, `AtomicInteger` and `AtomicLong` are the most important. Table 14-6 lists important methods in the `AtomicInteger` class. (The methods in `AtomicLong` are analogous to these.)

Table 14-6. *Important Methods in the AtomicInteger Class*

Method	Description
`AtomicInteger()`	Creates an instance of `AtomicInteger` with initial value 0.
`AtomicInteger(int initVal)`	Creates an instance of `AtomicInteger` with initial value `initVal`.
`int get()`	Returns the integer value held in this object.
`void set(int newVal)`	Resets the integer value held in this object to `newVal`.
`int getAndSet(int newValue)`	Returns the current `int` value held in this object and sets the value held in this object to `newVal`.
`boolean compareAndSet (int expect, int update)`	Compares the `int` value of this object to the `expect` value, and if they are equal, sets the `int` value of this object to the `update` value.
`int getAndIncrement()`	Returns the current value of the integer value in this object and increments the integer value in this object. Similar to the behavior of `i++` where `i` is an `int`.
`int getAndDecrement()`	Returns the current value of the integer value in this object and decrements the integer value in this object. Similar to the behavior of `i--` where `i` is an `int`.
`int getAndAdd(int delta)`	Returns the integer value held in this object and adds given delta value to the integer value.
`int incrementAndGet()`	Increments the current value of the integer value in this object and returns that value. Similar to the behavior of `++i` where `i` is an `int`.
`int decrementAndGet()`	Decrements the current integer value in this object and returns that value. Similar to behavior of `--i` where `i` is an `int`.
`int addAndGet(int delta)`	Adds the `delta` value to the current value of the integer in this object and returns that value.

`int intValue()`
`long longValue()`
`float floatValue()`
`doubleValue()`
	Casts the current `int` value of the object and returns it as `int`, `long`, `float`, or `double` values.

Let's try out an example to understand how to use AtomicInteger or AtomicLong. Assume that you have a counter value that is public and accessible by all threads. How do you update or access this common counter value safely without introducing the data race problem (discussed in the previous chapter)? Obviously, you can use the synchronized keyword to ensure that the critical section (the code that modifies the counter value) is accessed by only one thread at a given point in time. The critical section will be very small, as in

```
public void run() {
        synchronized(SharedCounter.class) {
                SharedCounter.count++;
        }
}
```

However, this code is inefficient since it acquires and releases the lock every time just to increment the value of count. Alternatively, if you declare count as AtomicInteger or AtomicLong (whichever is suitable), then there is no need to use a lock with synchronized keyword. Listing 14-8 gives the full program to show how to use AtomicLong in practice.

Listing 14-8. AtomicVariableTest.java

```
import java.util.concurrent.atomic.*;

// Class to demonstrate how incrementing "normal" (i.e., thread unsafe) integers and incrementing
// "atomic" (i.e., thread safe) integers are different: Incrementing a shared Integer object without
locks can result
// in a data race; however, incrementing a shared AtomicInteger will not result in a data race.

class AtomicVariableTest {
        // Create two integer objects – one normal and another atomic –  with same initial value
        private static Integer integer = new Integer(0);
        private static AtomicInteger atomicInteger = new AtomicInteger(0);

        static class IntegerIncrementer extends Thread {
                public void run() {
                        System.out.println("Incremented value of integer is: " + ++integer);
                }
        }
        static class AtomicIntegerIncrementer extends Thread {
                public void run() {
                        System.out.println("Incremented value of atomic integer is: "
                                        + atomicInteger.incrementAndGet());
                }
        }
        public static void main(String []args) {
                // create three threads each for incrementing atomic and "normal" integers
                for(int i = 0; i < 5; i++) {
                        new IntegerIncrementer().start();
                        new AtomicIntegerIncrementer().start();
                }
        }
}
```

The actual output depends on thread scheduling. In one run it printed the following:

```
Incremented value of atomic integer is: 1
Incremented value of integer is: 1
Incremented value of integer is: 1
Incremented value of atomic integer is: 2
Incremented value of integer is: 2
Incremented value of atomic integer is: 3
Incremented value of integer is: 3
Incremented value of integer is: 4
Incremented value of atomic integer is: 4
Incremented value of atomic integer is: 5
```

In this output, notice that incrementing the `Integer` object has resulted in a data race: the final value of `Integer` after incrementing it 5 times (from initial value 0) is 4. For `AtomicInteger`, however, it is 5—which is correct.

Let's analyze this program. The `AtomicVariableTest` has two data members—one of type `Integer` and the other of type `AtomicInteger`—with same initial value.

There are two `Thread` classes. One class increments `Integer` value in its `run()` method, and the other increments `AtomicInteger` in its `run()` method. In the `main()` method, you spawn five threads of these two kind of `Threads`. The output shows that incrementing the `Integer` value is prone to a data race when it is without a lock, whereas it is safe to increment the `AtomicInteger` value without any locks.

Locks

In the last chapter, we discussed the `synchronized` keyword and how it enforces that only one thread executes in a critical section at a time. The `java.util.concurrent.locks` package provides facilities that are more sophisticated. In this section, we will discuss the `Lock` interface.

Using a `Lock` object is similar to obtaining implicit locks using the `synchronized` keyword. The aim of both constructs is the same: to ensure that only one thread accesses a shared resource at a time. However, unlike the `synchronized` keyword, `Locks` also support the wait/notify mechanism along with its support for `Condition` objects.

You can think of using `synchronized` for locking implicitly and using `Lock` objects for locking explicitly.

The advantage of using the `synchronized` keyword (implicit locking) is that you don't have to remember to release the lock in a `finally` block since, at the end of the `synchronized` block (or method), code will be generated to automatically release the lock. Although this is a useful feature, there are some situations where you may need to control the release of the lock manually (say, for releasing it other than at the end of that block), and `Lock` objects provide this flexibility. However, it is your responsibility to ensure that you release the lock in a `finally` block while using `Lock` objects. The following snippet describes the usage idiom for a `Lock`:

```
Lock lock = /* get Lock type instance */;
lock.lock();
```

```
try {
       // critical section
}
finally {
       lock.unlock();
}
```

Another difference between implicit locks and explicit `Lock` objects is that you can do a "non-blocking attempt" to acquire locks with `Locks`. Well, what does "non-blocking attempt" mean here? You get a lock if that lock is available for locking, or you can back out from requesting the lock using the `tryLock()` method on a `Lock` object. Isn't it interesting? If you acquire the lock successfully, then you can carry out the task to be carried out in a critical section; otherwise you execute an alternative action. It is noteworthy that an overloaded version of the `tryLock()` method takes the timeout value as an argument so that you can wait to acquire the lock for the specified time.

```
tryLock(long time, TimeUnit unit).
```

With `tryLock()`, the idiom to use the `Lock` object is:

```
Lock lock = /* get Lock type instance */;
if(tryLock()) {
        try {
                // critical section
        }
        finally {
                lock.unlock();
        }
}
else {

}
```

Using `tryLock()` helps avoid some of the thread synchronization-related problems discussed in the last chapter, such as deadlocks and livelocks. Table 14-7 lists important methods in the `Lock` class.

Table 14-7. *Important Methods in the Lock Class*

Method	Description
`void lock()`	Acquires the lock.
`boolean tryLock()`	Acquires the lock and returns `true` if the lock is available; if the lock is not available, it does not acquire the lock and returns `false`.
`boolean tryLock(long time, TimeUnit unit)`	Same as the previous method `tryLock()`, but waits for the given waiting time before failing to acquire the lock and returns `false`.
`void lockInterruptibly()`	Acquires a lock; during the process of a acquiring the lock, if another thread interrupts it, this method throws an `InterruptedException`
`Condition newCondition()`	Returns a `Condition` object associated with this `Lock` object.
`void unlock()`	Releases the lock.

Let's look at an example of a Lock object. In this example, you use a Lock object and pass it to threads to synchronize them on this Lock object. This program is a simple variation of the program using Semaphores given in Listing 14-1. In Listing 14-9, you simulate accessing an ATM machine, which is a shared resource. Of course, only one person can use an ATM machine at a time, hence the code for accessing the machine is a critical section.

Listing 14-9. ATMRoom.java

```
import java.util.concurrent.locks.*;

// This class simulates a situation where only one ATM machine is available and
// and five people are waiting to access the machine. Since only one person can
// access an ATM machine at a given time, others wait for their turn
class ATMMachine {
        public static void main(String []args) {
                // A person can use a machine again, and hence using a "reentrant lock"
                Lock machine = new ReentrantLock();

                // list of people waiting to access the machine
                new Person(machine, "Mickey");
                new Person(machine, "Donald");
                new Person(machine, "Tom");
                new Person(machine, "Jerry");
                new Person(machine, "Casper");
        }
}

// Each Person is an independent thread; their access to the common resource
// (the ATM machine in this case) needs to be synchronized using a lock
class Person extends Thread {
        private Lock machine;
        public Person(Lock machine, String name) {
                this.machine = machine;
                this.setName(name);
                this.start();
        }
        public void run() {
                try {
                        System.out.println(getName() + " waiting to access an ATM machine");
                        machine.lock();
                        System.out.println(getName() + " is accessing an ATM machine");
                        Thread.sleep(1000); // simulate the time required for withdrawing amount
                } catch(InterruptedException ie) {
                        System.err.println(ie);
                }
                finally {
                        System.out.println(getName() + " is done using the ATM machine");
                        machine.unlock();
                }
        }
}
```

Here is the output of this program:

```
Donald waiting to access an ATM machine
Jerry waiting to access an ATM machine
Tom waiting to access an ATM machine
Mickey waiting to access an ATM machine
Donald is accessing an ATM machine
Casper waiting to access an ATM machine
Donald is done using the ATM machine
Jerry is accessing an ATM machine
Jerry is done using the ATM machine
Tom is accessing an ATM machine
Tom is done using the ATM machine
Mickey is accessing an ATM machine
Mickey is done using the ATM machine
Casper is accessing an ATM machine
Casper is done using the ATM machine
```

As you can observe from the output, the machine is accessed by only one person at a time, though there may be others waiting to access it. In this program, the class `ATMMachine` creates a `Lock` object representing an ATM machine. There are five people waiting to access the machine, which is simulated by creating five instances of the `Person` class. The `Person` class extends the `Thread` and remembers the `Lock` object on which it has to acquire and release the lock.

The `run()` method simply acquires the lock, accesses the shared resource, and releases the lock in a `finally` block. The `Lock` object (machine variable here) ensures that only one thread accesses it at a given point in time. Other threads block while one thread is accessing the lock.

Note that you may get a different order of people accessing the machine if you try running this program. This is because the access order depends on how the scheduler in the JVM schedules the threads to run.

The `ReadWriteLock` interface (which extends from the `Lock` interface) specifies a lock that provides separate locks for read-only access and write access. You can use the `readLock()` and `writeLock()` methods to get instances of read and write locks, respectively. The `ReentrantReadWriteLock` class implements the `ReadWriteLock` interface.

Conditions

A `Condition` supports thread notification mechanism. When a certain condition is not satisfied, a thread can wait for another thread to satisfy that condition; that other thread could notify once the condition is met. A `condition` is bound to a `lock`. A `Condition` object offers three methods to support wait/notify pattern: `await()`, `signal()`, and `signalAll()`. These three methods are analogous to the `wait()`, `notify()`, and `notifyAll()` methods supported by the `Object` class.

A thread can wait for a condition to be true using the `await()` method, which is an interruptible blocking call. If you want non-interruptible waiting, you can call `awaitUninterruptibly()`. You can also specify time duration for the waiting using one of the overloaded methods:

- `long awaitNanos(long nanosTimeout)`
- `boolean await(long time, TimeUnit unit)`
- `boolean awaitUntil(Date deadline)`

Now let's look at an example that makes use of `Condition` objects. Assume that you're waiting for a person named Joe to come on train IC1122, which is from Madrid to Paris. When Joe's train arrives at the station, he informs you; you pick him up and go home.

Assuming that multiple trains can arrive at a railway station, you need to wait for a specific train to arrive. Once the train arrives that you're interested in, you get a "notification" or "signal" from that train. This scenario is a good candidate for using the wait/notify pattern. There are two ways to implement this pattern. The first option is to use implicit locks and make use of the `wait()` and `notifyAll()` methods in the `Object` class. The second option—shown in Listing 14-10—is to use the explicit `Lock` and `Condition` objects and use the `await()` and `signalAll()` methods in the `Condition` object.

Listing 14-10. RailwayStation.java

```
import java.util.concurrent.locks.*;

// This class simulates arrival of trains in a railway station.
class RailwayStation {
        // A common lock for synchronization
        private static Lock station = new ReentrantLock();
        // Condition to wait or notify the arrival of Joe in the station
        private static Condition joeArrival = station.newCondition();

        // Train class simulates arrival of trains independently
        static class Train extends Thread {
                public Train(String name) {
                    this.setName(name);
                }
                public void run() {
                        station.lock();
                        try {
                                System.out.println(getName() + ": I've arrived in station ");
                                if(getName().startsWith("IC1122")) {
                                        // Joe is coming in train IC1122 - he announces it to us
                                        joeArrival.signalAll();
                                }
                        }
                        finally {
                                station.unlock();
                        }
                }
        }

        // Our wait in the railway station for Joe is simulated by this thread. Once we get
notification from Joe
        // that he has arrived, we pick-him up and go home
        static class WaitForJoe extends Thread {
                public void run() {
                        System.out.println("Waiting in the station for IC1122 in which Joe is coming");
                        station.lock();
                        try {
                                // await Joe's train arrival
                                joeArrival.await();
```

```
                        // if this statement executes, it means we got a train arrival signal
                        System.out.println("Pick up Joe and go home");
                } catch(InterruptedException ie) {
                        ie.printStackTrace();
                }
                finally {
                        station.unlock();
                }
        }
    }

    // first create a thread that waits for Joe to arrive and then create new Train threads
    public static void main(String []args) throws InterruptedException {
            // we are waiting before the trains start coming
            new WaitForJoe().start();
            // Trains are separate threads - they can arrive in any order
            new Train("IC1234 - Paris to Munich").start();
            new Train("IC2211 - Paris to Madrid").start();
            new Train("IC1122 - Madrid to Paris").start();
            new Train("IC4321 - Munich to Paris").start();
    }
}
```

Here is the output of this program:

```
Waiting in the station for IC1122 in which Joe is coming
IC1234 - Paris to Munich: I've arrived in station
IC1122 - Madrid to Paris: I've arrived in station
IC2211 - Paris to Madrid: I've arrived in station
Pick up Joe and go home
IC4321 - Munich to Paris: I've arrived in station
```

Let's analyze how this program works. In the `RailwayStation` class you have a common `Lock` object named `station`. From that `station` object, you obtain a `Condition` object (remember that a condition is always associated with a lock) named `joeArrival`. You used the `newCondition()` method, so the resulting `Condition` object is an interruptible condition; you have not specified any time-out, so the awaiting thread will wait forever until it gets the signal.

The `Train` class is a `Thread` that simulates arrival of a train in the railway station. The `run()` method in `Train` first obtains the lock before announcing that the train has arrived, and it releases before the method exits. Note that if you call `await()` on the `Condition` object without acquiring a lock, you'll get an `IllegalMonitorStateException`. In the `run()` method, if the `Train` name is `IC1122`, it will signal us that Joe has arrived by calling `joeArrival.signalAll();`.

Your wait in the railway station for `Joe` is simulated by this `WaitForJoe` thread. In the `run()` method, you acquire the lock and wait for the `joeArrival` condition to be signaled. Once you are notified (i.e., signaled) that he has arrived, you pick him up and go home.

In multithreading, a common need is to wait for a condition to be satisfied by one thread before another thread can proceed. Using polling (i.e., repeatedly checking for a condition using a `while` loop) is a bad solution because this solution wastes CPU cycles; further, it is also prone to data races. Use guarded blocks using wait/notify or await/signal instead.

Multiple Conditions on a Lock

From the OCPJP 7 exam perspective, it is important to understand locks and conditions. So, we'll discuss one more detailed example that makes use of locks and conditions. In this program, we show how you can get multiple `Condition` objects on a `Lock` object.

Assume that you are asked to implement a fixed-size queue with the size of the queue determined at the time of thread creation. In a typical queue, if there are no elements in the queue and if the `remove()` method is called, it will throw a `NoSuchElementException` (as you saw in Listing 14-6). However, in this case, you want the thread to block until some other thread inserts an element. Similarly, if you try inserting in a queue that is already full, instead of throwing `IllegalStateException` to indicate that it is not possible to insert any more elements, the thread should block until an element is removed. In other words, you need to implement a simple blocking queue (see Listing 14-11).

Listing 14-11. BlockerQueue.java

```
import java.util.concurrent.locks.*;

// this implements a fixed size queue with size determined at the time of creation. I/ if remove()
is called
// when there are no elements, then the queue blocks (i.e., waits) until an element is inserted.
// If insert() is called when the queue is full, then the queue blocks until an element is removed

class BlockerQueue {
        // remember the max size of the queue
        private int size = 0;

        // array to store the elements in the queue
        private Object elements[];

        // pointer that points to the current element in the queue
        private int currPointer = 0;

        // internal lock used for synchronized access to the BlockerQueue
        private Lock internalLock = new ReentrantLock();

        // condition to wait for when queue is empty that makes use of the common lock
        private Condition empty = internalLock.newCondition();

        // condition to wait for when queue is full that makes use of the common lock
        private Condition full =  internalLock.newCondition();

        public BlockerQueue(int size) {
                this.size = size;
                elements = new Object[size];
        }

        // remove an element if available; or if there are no elements in the queue,
        // await insertion of an element. Once an element is inserted, notify to any threads
        // waiting for insertion in a full queue
        public Object remove() {
                Object element = null;
                internalLock.lock();
```

```
                try {
                        if(currPointer == 0) {
                                System.out.println("In remove(): no element to remove, so waiting
for insertion");
                                // cannot remove - no elements in the queue;
                                // so block until an element is inserted
                                empty.await();
                                // if control reaches here, that means some thread completed
                                // calling insert(), so proceed to remove that element
                                System.out.println("In remove(): got notification that an element has
got inserted");
                        }
                        // decrement the currPointer and then get the element
                        element = elements[--currPointer];
                        System.out.println("In remove(): removed the element " + element);

                        // an element is removed, so there is space for insertion
                        // so notify any threads waiting to insert
                        full.signalAll();
                        System.out.println("In remove(): signalled that there is space for insertion");
                } catch(InterruptedException ie) {
                        ie.printStackTrace();
                } finally {
                        internalLock.unlock();
                }
                return element;
        }

        // insert an element if there is space for insertion. if queue is full,
        // await for remove() to be called and get signal to proceed for insertion.
        // after insertion, signal any awaiting threads in case of an empty queue.
        public void insert(Object element) {
                internalLock.lock();
                try {
                        if(currPointer == size) {
                                System.out.println("In insert(): queue is full, so waiting for removal");
                                // cannot insert - the queue is full;
                                // so block until an element is removed
                                full.await();
                                // if control reaches here, that means some thread completed
                                // calling remove(), so proceed to insert this element
                                System.out.println("In insert(): got notification that remove got called,
so proceeding to insert the element");
                        }
                        // get the element and after that decrement the currPointer
                        elements[currPointer++] = element;
                        System.out.println("In insert(): inserted the element " + element);
                        // an element is inserted, so any other threads can remove it...
                        // so notify any threads waiting to remove
                        empty.signalAll();
                        System.out.println("In insert(): notified that queue is not empty");
```

```
                } catch(InterruptedException ie) {
                        ie.printStackTrace();
                } finally {
                        internalLock.unlock();
                }
        }
}
```

Here is test code for this class:

```
class BlockerQueueTest1 {
        public static void main(String []args) {
                final BlockerQueue blockerQueue = new BlockerQueue(2);
                new Thread() {
                        public void run() {
                                System.out.println("Thread1: attempting to remove an item from the queue ");
                                Object o = blockerQueue.remove();
                        }
                }.start();

                new Thread() {
                        public void run() {
                                System.out.println("Thread2: attempting to insert an item to the queue");
                                blockerQueue.insert("one");
                        }
                }.start();
       }
}
```

This test code prints the following:

```
Thread1: attempting to remove an item from the queue
In remove(): no element to remove, so waiting for insertion
Thread2: attempting to insert an item to the queue
In insert(): inserted the element one
In insert(): notified that queue is not empty
In remove(): got notification that an element has got inserted
In remove(): removed the element one
In remove(): signalled that there is space for insertion
```

As you can see from the output, the `remove()` method got called first, which waits for `insert()` to be called. Once `insert()` is complete, the `remove()` method successfully removes the element from the queue. Now, let's try another test case to test if blocking in the `insert()` method works:

```
class BlockerQueueTest2 {
        public static void main(String []args) {
                final BlockerQueue blockerQueue = new BlockerQueue(3);
                blockerQueue.insert("one");
                blockerQueue.insert("two");
                blockerQueue.insert("three");
                new Thread() {
```

```
                        public void run() {
                                System.out.println("Thread2: attempting to insert an item to the queue");
                                blockerQueue.insert("four");
                        }
                }.start();

                new Thread() {
                        public void run() {
                                System.out.println("Thread1: attempting to remove an item from the queue ");
                                Object o = blockerQueue.remove();
                        }
                }.start();
        }
}
```

This test code prints the following:

```
In insert(): inserted the element one
In insert(): notified that queue is not empty
In insert(): inserted the element two
In insert(): notified that queue is not empty
In insert(): inserted the element three
In insert(): notified that queue is not empty
Thread2: attempting to insert an item to the queue
In insert(): queue is full, so waiting for removal
Thread1: attempting to remove an item from the queue
In remove(): removed the element three
In remove(): signalled that there is space for insertion
In insert(): got notification that remove got called, so proceeding to insert the element
In insert(): inserted the element four
In insert(): notified that queue is not empty
```

As you can see from the output, when a thread invokes `insert` on the full queue (you have specified the capacity as 3 elements in this case), the thread blocks. Once another thread removed an element from the queue, the blocked thread resumes and successfully inserts the element.

Use Executors and ThreadPools

You can directly create and manage threads in the application by creating `Thread` objects. However, if you want to abstract away the low-level details of multi-threaded programming, you can make use of the `Executor` interface.

Figure 14-1 shows the important classes and interfaces in the `Executor` hierarchy. In this section, you'll focus on using the `Executor` interface, `ExecutorService`, and `ThreadPools`. We'll cover `ForkJoinPool` in the next section, "Use the Parallel Fork/Join Framework."

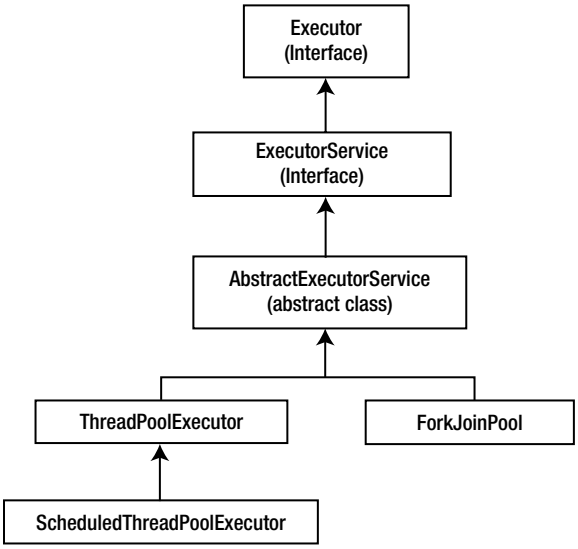


Figure 14-1. *Important classes/interfaces in the Executor hierarchy*

Executor

`Executor` is an interface that declares only one method: `void execute(Runnable)`. This may not look like a big interface by itself, but its derived classes (or interfaces), such as `ExecutorService`, `ThreadPoolExecutor`, and `ForkJoinPool`, support useful functionality. We will discuss the derived classes of `Executor` in more detail in the rest of this section. For now, look at Listing 14-12 for a simple example of the `Executor` interface to understand how to implement this interface and use it in practice.

Listing 14-12. ExecutorTest.java

```
import java.util.concurrent.*;

// This Task class implements Runnable, so its a Thread object
class Task implements Runnable {
        public void run() {
                System.out.println("Calling Task.run() ");
        }
}

// This class implements Executor interface and should override execute(Runnable) method.
// We provide an overloaded execute method with an additional argument 'times' to create and
// run the threads for given number of times
class RepeatedExecutor implements Executor {
        public void execute(Runnable runnable) {
                new Thread(runnable).start();
        }
```

```
        public void execute(Runnable runnable, int times) {
                System.out.printf("Calling Task.run() thro' Executor.execute() for %d times %n", times);
                for(int i = 0; i < times; i++) {
                        execute(runnable);
                }
        }
}

// This class spawns a Task thread and explicitly calls start() method.
// It also shows how to execute a Thread using Executor
class ExecutorTest {
        public static void main(String []args) {
                Runnable runnable = new Task();
                System.out.println("Calling Task.run() by directly creating a Thread object");
                Thread thread = new Thread(runnable);
                thread.start();
                RepeatedExecutor executor = new RepeatedExecutor();
                executor.execute(runnable, 3);
        }
}
```

Here is the output of this program:

```
Calling Task.run() by directly creating a Thread object
Calling Task.run()
Calling Task.run() thro' Executor.execute() for 3 times
Calling Task.run()
Calling Task.run()
Calling Task.run()
```

In this program, you have a `Task` class that implements `Runnable` by providing the definition of the `run()` method. The class `RepeatedExecutor` implements the `Executor` interface by providing the definition of the `execute(Runnable)` method.

Both `Runnable` and `Executor` are similar in the sense that they provide a single method for implementation. In this definition you may have noticed that `Exectutor` by itself is not a thread, and you must create a `Thread` object to execute the `Runnable` object passed in the `execute()` method. However, the main difference between `Runnable` and `Exectutor` is that `Executor` is meant to abstract how the thread is executed. For example, depending on the implementation of `Executor`, `Exectutor` may schedule a thread to run at a certain time, or execute the thread after a certain delay period.

In this program, you have overloaded the `execute()` method with an additional argument to create and execute threads a certain number of times. In the `main()` method, you first create a `Thread` object and schedule it for running. After that, you instantiate `RepeatedExectutor` to execute the thread three times.

Callable, Executors, ExecutorService, ThreadPool, and Future

`Callable` is an interface that declares only one method: `call()`. Its full signature is `V call() throws Exception`. It represents a task that needs to be completed by a thread. Once the task completes, it returns a value. For some reason, if the `call()` method cannot execute or fails, it throws an `Exception`.

To execute a task using the `Callable` object, you first create a thread pool. A thread pool is a collection of threads that can execute tasks. You create a thread pool using the `Executors` utility class. This class provides methods to get instances of thread pools, thread factories, etc.

The ExecutorService interface implements the Executor interface and provides services such as termination of threads and production of Future objects. Some tasks may take considerable execution time to complete. So, when you submit a task to the executor service, you get a Future object.

Future represents objects that contain a value that is returned by a thread in the future (i.e., it returns the value once the thread terminates in the "future"). You can use the isDone() method in the Future class to check if the task is complete and then use the get() method to fetch the task result. If you call the get() method directly while the task is not complete, the method blocks until it completes and returns the value once available.

Enough talking—try a simple example to see how these classes work together (Listing 14-13).

Listing 14-13. CallableTest.java

```
// Factorial implements Callable so that it can be passed to a ExecutorService
// and get executed as a task.
class Factorial implements Callable<Long> {
        long n;
        public Factorial(long n) {
                this.n = n;
        }
        public Long call() throws Exception {
                if(n <= 0) {
                        throw new Exception("for finding factorial, N should be > 0");
                }
                long fact = 1;
                for(long longVal = 1; longVal <= n; longVal++) {
                        fact *= longVal;
                }
                return fact;
        }
}

// Illustrates how Callable, Executors, ExecutorService, and Future are related;
// also shows how they work together to execute a task
class CallableTest {
        public static void main(String []args) throws Exception {
                // the value for which we want to find the factorial
                long N = 20;
                // get a callable task to be submitted to the executor service
                Callable<Long> task = new Factorial(N);
                // create an ExecutorService with a fixed thread pool consisting of one thread
                ExecutorService es = Executors.newSingleThreadExecutor();
                // submit the task to the executor service and store the Future object
                Future<Long> future = es.submit(task);
                // wait for the get() method that blocks until the computation is complete.
                System.out.printf("factorial of %d is %d", N, future.get());
                // done. shutdown the executor service since we don't need it anymore
                es.shutdown();
        }
}
```

The program prints the following:

```
factorial of 20 is 2432902008176640000
```

In this program, you have a `Factorial` class that implements `Callable`. Since the task is to compute the factorial of a number `N`, the task needs to return a result. You use `Long` type for the factorial value, so you implement `Callable<Long>`. Inside the `Factorial` class, you define the `call()` method that actually performs the task (the task here is to compute the factorial of the given number). If the given value `N` is negative or zero, you don't perform the task and throw an exception to the caller. Otherwise, you loop from 1 to `N` and find the factorial value.

In the `CallableTest` class, you first create an instance of the `Factorial` class. You then need to execute this task. For the sake of simplicity, you get a singled-threaded executor by calling the `newSingleThreadExecutor()` method in the `Executors` class. Note that you could use other methods such as `newFixedThreadPool(nThreads)` to create a thread pool with multiple threads depending on the level of parallelism you need.

Once you get an `ExecutorService`, you submit the task for execution. `ExecutorService` abstracts details such as when the task is executed, how the task is assigned to the threads, etc. You get a reference to `Future<Long>` when you call the `submit(task)` method. From this future reference, you call the `get()` method to fetch the result after completing the task. If the task is still executing when you call `future.get()`, this `get()` method will block until the task execution completes. Once the execution is complete, you need to manually release the `ExecutorService` by calling the `shutdown()` method.

Now that you are familiar with the basic mechanism of how to execute tasks, here's a complex example. Assume that your task is to find the sum of numbers from 1 to N where N is a large number (a million in our case). Of course, you can use the formula `[(N * (N + 1)) / 2]` to find out the sum. Yes, you'll make use of this formula to check if the summation from 1…N is correct or not. However, just for illustration, you'll divide the range 1 to 1 million to N sub-ranges and by spawn N threads to sum up numbers in that sub-range; see Listing 14-14.

Listing 14-14. SumOfN.java

```
import java.util.*;
import java.util.concurrent.*;

// We create a class SumOfN that sums the values from 1..N where N is a large number.
// We divide the task
// to sum the numbers to 10 threads (which is an arbitrary limit just for illustration).
// Once computation is complete, we add the results of all the threads,
// and check if the calculation is correct by using the formula (N * (N + 1))/2.
class SumOfN {
        private static long N = 1_000_000L;    // one million
        private static long calculatedSum = 0; // value to hold the sum of values in range 1..N
        private static final int NUM_THREADS = 10; // number of threads to create for distributing the effort

        // This Callable object sums numbers in range from..to
        static class SumCalc implements Callable<Long> {
                long from, to, localSum = 0;

                public SumCalc(long from, long to) {
                        this.from = from;
                        this.to = to;
                }
                public Long call() {
                        // add in range 'from' .. 'to' inclusive of the value 'to'
                        for(long i = from; i <= to; i++) {
                                localSum += i;
                        }
                        return localSum;
                }
        }
```

```
        // In the main method we implement the logic to divide the summation tasks to
        // given number of threads and finally check if the calculated sum is correct
        public static void main(String []args) {
                // Divide the task among available fixed number of threads
                ExecutorService executorService = Executors.newFixedThreadPool(NUM_THREADS);
                // store the references to the Future objects in a List for summing up together
                List<Future<Long>> summationTasks = new ArrayList<>();
                long nByTen = N/10;       // divide N by 10 so that it can be submitted as 10 tasks
                for(int i = 0; i < NUM_THREADS; i++) {
                        // create a summation task
                        // starting from (10 * 0) + 1 .. (N/10 * 1) to (10 * 9) + 1 .. (N/10 * 10)
                        long fromInInnerRange = (nByTen * i) + 1;
                        long toInInnerRange = nByTen * (i+1);
                        System.out.printf("Spawning thread for summing in range %d to %d %n",
fromInInnerRange, toInInnerRange);
                        // Create a callable object for the given summation range
                        Callable<Long> summationTask =
                               new SumCalc(fromInInnerRange, toInInnerRange);
                        // submit that task to the executor service
                        Future<Long> futureSum = executorService.submit(summationTask);
                        // it will take time to complete, so add it to the list to revisit later
                        summationTasks.add(futureSum);
                }

                // now, find the sum from each task
                for(Future<Long> partialSum : summationTasks) {
                        try {
                                // the get() method will block (i.e., wait) until the computation is over
                                calculatedSum += partialSum.get();
                        } catch(CancellationException | ExecutionException
                                | InterruptedException exception) {
                                // unlikely that you get an exception - exit in case something goes wrong
                                exception.printStackTrace();
                                System.exit(-1);
                        }
                }

                // now calculate the sum using formula (N * (N + 1))/2 without doing the hard-work
                long formulaSum = (N * (N + 1))/2;
                // print the sum using formula and the ones calculated one by one
                // they must be equal!
                System.out.printf("Sum by threads = %d, sum using formula = %d",
                        calculatedSum, formulaSum);
        }
}
```

Here is the output of this program:

```
Spawning thread for summing in range 1 to 100000
Spawning thread for summing in range 100001 to 200000
Spawning thread for summing in range 200001 to 300000
Spawning thread for summing in range 300001 to 400000
```

```
Spawning thread for summing in range 400001 to 500000
Spawning thread for summing in range 500001 to 600000
Spawning thread for summing in range 600001 to 700000
Spawning thread for summing in range 700001 to 800000
Spawning thread for summing in range 800001 to 900000
Spawning thread for summing in range 900001 to 1000000
Sum by threads = 500000500000, sum using formula = 500000500000
```

Let's now analyze how this program works. In this program, you need to find the sum of 1..N where N is one million (a large number). The class `SumCalc` implements `Callable<Long>` to sum the values in the range `from` to `to`. The `call()` method performs the actual computation of the sum by looping from `from` to `to` and returns the intermediate sum value as a `Long` value.

In this program, you divide the summation task among multiple threads. You can determine the number of threads based on the number of cores available in your processor; however, for the sake of keeping the program simpler, use ten threads.

In the `main()` method, you create a `ThreadPool` with ten threads. You are going to create ten summation tasks, so you need a container to hold the references to those tasks. Use `ArrayList` to hold the `Future<Long>` references.

In the first `for` loop in `main()`, you create ten tasks and submit them to the `ExecutorService`. As you submit a task, you get a `Future<Long>` reference and you add it to the `ArrayList`.

Once you've created the ten tasks, you traverse the array list in the next `for` loop to get the results of the tasks. You sum up the partial results of the individual tasks to compute the final sum.

Once you get the computed sum of values from one to one million, you use the simple formula `N * (N + 1)/2` to find the formula sum. From the output, you can see that the computed sum and the formula sum are equal, so you can ascertain that your logic of dividing the tasks and combining the results of the tasks worked correctly.

Now, before we move on to discuss the fork/join framework, we'll quickly discuss a few classes that are useful for concurrent programming.

ThreadFactory

`ThreadFactory` is an interface that is meant for creating threads instead of explicitly creating threads by calling new `Thread()`. For example, assume that you often create high-priority threads. You can create a `MaxPriorityThreadFactory` to set the default priority of threads created by that factory to maximum priority (see Listing 14-15).

Listing 14-15. TestThreadFactory.java

```
import java.util.concurrent.*;

// A ThreadFactory implementation that sets the thread priority to max
// for all the threads it creates
class MaxPriorityThreadFactory implements ThreadFactory {
        private static long count = 0;
        public Thread newThread(Runnable r) {
             Thread temp = new Thread(r);
             temp.setName("prioritythread" + count++);
             temp.setPriority(Thread.MAX_PRIORITY);
             return temp;
        }
}
```

```
class ARunnable implements Runnable {
        public void run() {
                System.out.println("Running the created thread ");
        }
}

class TestThreadFactory {
        public static void main(String []args) {
                ThreadFactory threadFactory = new MaxPriorityThreadFactory();
                Thread t1 = threadFactory.newThread(new ARunnable());
                System.out.println("The name of the thread is " + t1.getName());
                System.out.println("The priority of the thread is " + t1.getPriority());
                t1.start();
        }
}
```

It prints the following:

```
The name of the thread is prioritythread0
The priority of the thread is 10
Running the created thread
```

With the use of `ThreadFactory`, you can reduce boilerplate code to set thread priority, name, thread-pool, etc.

The ThreadLocalRandom Class

When you do concurrent programming, you'll find that there is often a need to generate random numbers. Using `Math.random()` is not efficient for concurrent programming. For this reason, the `java.util.concurrent` package introduces the `ThreadLocalRandom` class, which is suitable for use in concurrent programs. You can use `ThreadLocalRandom.current()` and then call methods such as `nextInt()` and `nextFloat()` to generate the random numbers.

TimeUnit Enumeration

You've already seen some methods earlier in this chapter that take `TimeUnit` as an argument. `TimeUnit` is an enumeration that is used to specify the resolution of the timing. The unit of time in `TimeUnit` can be one of `DAYS`, `HOURS`, `MINUTES`, `SECONDS`, `MICROSECONDS`, `MILLISECONDS`, or `NANOSECONDS`. The enumeration also has useful methods for converting between these time units. For example,

```
System.out.printf("One day has %d hours, %d minutes, %d seconds",
        TimeUnit.DAYS.toHours(1), TimeUnit.DAYS.toMinutes(1), TimeUnit.DAYS.toSeconds(1));
```

prints

```
One day has 24 hours, 1440 minutes, 86400 seconds
```

Some of the methods in the Java API use specific periods. For example, the `sleep()` method takes time to sleep in milliseconds. So, what if you want to specify the time for thread sleep in some other time unit, say seconds or days? `TimeUnit` makes this task easy. See Listing 14-16 for an example.

Listing 14-16. TimeUnitExample.java

```
import java.util.concurrent.TimeUnit;

// A simple example showing how to make use of TimeUnit enumeration
class TimeUnitExample {
        public static void main(String []args) throws InterruptedException {
                System.out.println("Calling sleep() method on main thread for 2 seconds");
                // Thread.sleep takes milli-seconds as argument. By using TimeUnit enumeration,
                // you can specify the time to sleep in other time units such as hours, minutes,
                // seconds, etc.
                Thread.sleep(TimeUnit.SECONDS.toMillis(2));
                System.out.println("main thread wakes up from sleep");
        }
}
```

Use the Parallel Fork/Join Framework

The Fork/Join framework in the `java.util.concurrent` package helps simplify writing parallelized code. The framework is an implementation of the `ExecutorService` interface and provides an easy-to-use concurrent platform in order to exploit multiple processors. This framework is very useful for modeling divide-and-conquer problems. This approach is suitable for tasks that can be divided recursively and computed on a smaller scale; the computed results are then combined. Dividing the task into smaller tasks is *forking*, and merging the results from the smaller tasks is *joining*.

The Fork/Join framework uses the work-stealing algorithm: when a `worker` thread completes its work and is free, it takes (or "steals") work from other threads that are still busy doing some work. Initially, it will appear to you that using Fork/Join is a complex task. Once you get familiar with it, however, you'll realize that it is conceptually easy and that it significantly simplifies your job. The key is to recursively subdivide the task into smaller chunks that can be processed by separate threads.

Briefly, the Fork/Join algorithm is designed as follows:

```
forkJoinAlgorithm() {
        split tasks;
        fork the tasks;
        join the tasks;
        compose the results;
}
```

Here is the pseudo-code of how these steps work:

```
doRecursiveTask(input) {
        if (the task is small enough to be handled by a thread) {
                compute the small task;
                if there is a result to return, do so
        }
        else {
                divide (i.e., fork) the task into two parts
                call compute() on first task, join() on second task, combine both results and return
        }
}
```

Figure 14-2 visualizes how the task is recursively subdivided into smaller tasks and how the partial results are combined. As shown by the figure, a task is split into two subtasks, and then each subtask is again split in two subtasks, and so on until each split subtask is computable by each thread. Once a thread completes the computation, it returns the result for combining it with other results; in this way all the computed results are combined back.

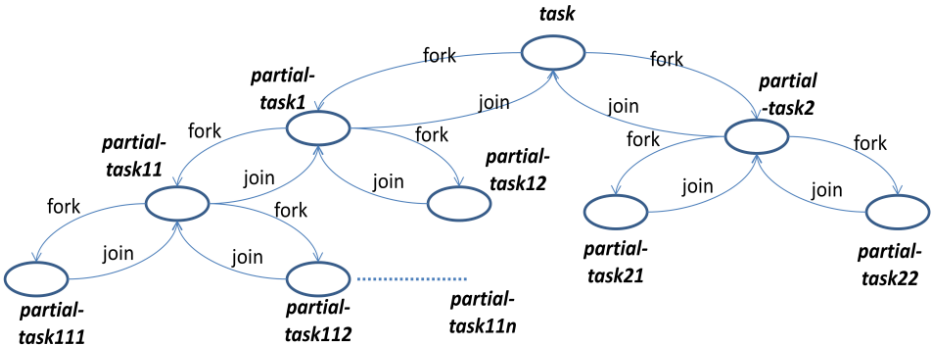


Figure 14-2. *How the Fork/Join framework uses divide-and-conquer to complete the task*

Useful Classes of the Fork/Join Framework

The following classes play key roles in the Fork/Join framework: `ForkJoinPool`, `ForkJoinTask`, `RecursiveTask`, and `RecursiveAction`. Let's consider these classes in more detail.

- `ForkJoinPool` is the most important class in the Fork/Join framework. It is a thread pool for running fork/join tasks—it executes an instance of `ForkJoinTask`. It executes tasks and manages their lifecycle. Table 14-8 lists the important methods belonging to this abstract class.

Table 14-8. *Important Methods in the ForkJoinPool Class*

Method	Description
`void execute(ForkJoinTask<?> task)`	Executes a given task asynchronously.
`<T> T invoke(ForkJoinTask<T> task)`	Executes the given task and returns the computed result.
`<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)`	Executes all the given tasks and returns a list of future objects when all the tasks are completed.
`boolean isTerminated()`	Returns `true` if all the tasks are completed.
`int getParallelism()`	Status checking methods.
`int getPoolSize()`	
`long getStealCount()`	
`int getActiveThreadCount()`	
`<T> ForkJoinTask<T> submit(Callable<T> task)`	Executes a submitted task. Overloaded versions take different types of tasks; returns a Task object or a Future object.
`<T> ForkJoinTask<T> submit(ForkJoinTask<T> task)`	
`ForkJoinTask<?> submit(Runnable task)`	
`<T> ForkJoinTask<T> submit(Runnable task, T result)`	

- `ForkJoinTask<V>` is a lightweight thread-like entity representing a task that defines methods such as `fork()` and `join()`. Table 14-9 lists the important methods of this class.

Table 14-9. *Important Methods in the ForkJoinTask Class*

Method	Description
`boolean cancel(boolean mayInterruptIfRunning)`	Attempts to cancel the execution of the task.
`ForkJoinTask<V> fork()`	Executes the task asynchronously.
`V join()`	Returns the result of the computation when the computation is done.
`V get()`	Returns the result of the computation; waits if the computation is not complete.
`V invoke()`	Starts the execution of the submitted tasks; waits until computation complete, and returns results.
`static <T extends ForkJoinTask<?>> Collection<T> invokeAll(Collection<T> tasks)`	
`boolean isCancelled()`	Returns `true` if the task is cancelled.
`boolean isDone()`	Returns `true` if the task is completed.

- `RecursiveTask<V>` is a task that can run in a `ForkJoinPool`; the `compute()` method returns a value of type V. It inherits from `ForkJoinTask`.
- `RecursiveAction` is a task that can run in a `ForkJoinPool`; its `compute()` method performs the actual computation steps in the task. It is similar to `RecursiveTask`, but does not return a value.

Using the Fork/Join Framework

Let's ascertain how you can use Fork/Join framework in problem solving. Here are the steps to use the framework:

- First, check whether the problem is suitable for the Fork/Join framework or not. Remember: the Fork/Join framework is not suitable for all kinds of tasks. This framework is suitable if your problem fits this description:
  - The problem can be designed as a recursive task where the task can be subdivided into smaller units and the results can be combined together.
  - The subdivided tasks are independent and can be computed separately without the need for communication between the tasks when computation is in process. (Of course, after the computation is over, you will need to join them together.)
- If the problem you want to solve can be modeled recursively, then define a task class that extends either `RecursiveTask` or `RecursiveAction`. If a task returns a result, extend from `RecursiveTask`; otherwise extend from `RecursiveAction`.
- Override the `compute()` method in the newly defined task class. The `compute()` method actually performs the task if the task is small enough to be executed; or split the task into subtasks and invoke them. The subtasks can be invoked either by `invokeAll()` or `fork()` method (use `fork()` when the subtask returns a value). Use the `join()` method to get the computed results (if you used `fork()` method earlier).

- Merge the results, if computed from the subtasks.
- Then instantiate `ForkJoinPool`, create an instance of the `task` class, and start the execution of the task using the `invoke()` method on the `ForkJoinPool` instance.
- That's it—you are done.

Now let's try solving the problem of how to sum 1..N where N is a large number. In Listing 14-16, you subdivided the sum computation task iteratively into ten sub-ranges; then you computed the sum for each sub-range and then computed the sum-of-the-partial sums. Alternatively, you can solve this problem rescursively using the Fork/Join framework (Listing 14-17).

Listing 14-17. SumOfNUsingForkJoin.java

```
import java.util.concurrent.*;

// This class illustrates how we can compute sum of 1..N numbers using fork/join framework.
// The range of numbers are divided into half until the range can be handled by a thread.
// Once the range summation completes, the result gets summed up together.

class SumOfNUsingForkJoin {
        private static long N = 1000_000; // one million - we want to compute sum
                                          // from 1 .. one million
        private static final int NUM_THREADS = 10; // number of threads to create for
                                                   // distributing the effort

        // This is the recursive implementation of the algorithm; inherit from RecursiveTask
        // instead of RecursiveAction since we're returning values.
        static class RecursiveSumOfN extends RecursiveTask<Long> {
                long from, to;
                // from and to are range of values to sum-up
                public RecursiveSumOfN(long from, long to) {
                        this.from = from;
                        this.to = to;
                }
                // the method performs fork and join to compute the sum.
                // if the range of values can be summed by a thread
                // (remember that we want to divide the summation task equally among NUM_THREADS)
                // then, sum the range of numbers from..to using a simple for loop
                // otherwise, fork the range and join the results
                public Long compute() {
                        if( (to - from) <= N/NUM_THREADS) {
                                // the range is something that can be handled by a thread, so do summation
                                long localSum = 0;
                                // add in range 'from' .. 'to' inclusive of the value 'to'
                                for(long i = from; i <= to; i++) {
                                        localSum += i;
                                }
                                System.out.printf("\t Summing of value range %d to %d is %d %n",
from,to, localSum);
                                return localSum;
                        }
```

```
                        else { // no, the range is big for a thread to handle, so fork the computation
                                // we find the mid-point value in the range from..to
                                long mid = (from + to)/2;
                                System.out.printf("Forking computation into two ranges: " +
                                        "%d to %d and %d to %d %n", from, mid, mid, to);
                                // determine the computation for first half with the range from..mid
                                RecursiveSumOfN firstHalf = new RecursiveSumOfN(from, mid);
                                // now, fork off that task
                                firstHalf.fork();
                                // determine the computation for second half with the range mid+1..to
                                RecursiveSumOfN secondHalf = new RecursiveSumOfN(mid + 1, to);
                                long resultSecond = secondHalf.compute();
                                // now, wait for the first half of computing sum to
                                // complete, once done, add it to the remaining part
                                return firstHalf.join() + resultSecond;
                        }
                }
        }

        public static void main(String []args) {
                // Create a fork-join pool that consists of NUM_THREADS
                ForkJoinPool pool = new ForkJoinPool(NUM_THREADS);
                // submit the computation task to the fork-join pool
                long computedSum = pool.invoke(new RecursiveSumOfN(0, N));
                // this is the formula sum for the range 1..N
                long formulaSum = (N * (N + 1)) / 2;
                // Compare the computed sum and the formula sum
                System.out.printf("Sum for range 1..%d; computed sum = %d, formula sum = %d %n", N,
computedSum, formulaSum);
        }
}
```

The program prints the following:

```
Forking computation into two ranges: 0 to 500000 and 500000 to 1000000
Forking computation into two ranges: 0 to 250000 and 250000 to 500000
Forking computation into two ranges: 0 to 125000 and 125000 to 250000
Forking computation into two ranges: 0 to 62500 and 62500 to 125000
         Summing of value range 0 to 62500 is 1953156250
         Summing of value range 62501 to 125000 is 5859406250
Forking computation into two ranges: 125001 to 187500 and 187500 to 250000
         Summing of value range 125001 to 187500 is 9765656250
         Summing of value range 187501 to 250000 is 13671906250
Forking computation into two ranges: 250001 to 375000 and 375000 to 500000
Forking computation into two ranges: 250001 to 312500 and 312500 to 375000
         Summing of value range 250001 to 312500 is 17578156250
         Summing of value range 312501 to 375000 is 21484406250
Forking computation into two ranges: 375001 to 437500 and 437500 to 500000
         Summing of value range 375001 to 437500 is 25390656250
         Summing of value range 437501 to 500000 is 29296906250
Forking computation into two ranges: 500001 to 750000 and 750000 to 1000000
Forking computation into two ranges: 500001 to 625000 and 625000 to 750000
```

```
Forking computation into two ranges: 500001 to 562500 and 562500 to 625000
         Summing of value range 500001 to 562500 is 33203156250
         Summing of value range 562501 to 625000 is 37109406250
Forking computation into two ranges: 625001 to 687500 and 687500 to 750000
         Summing of value range 625001 to 687500 is 41015656250
         Summing of value range 687501 to 750000 is 44921906250
Forking computation into two ranges: 750001 to 875000 and 875000 to 1000000
Forking computation into two ranges: 750001 to 812500 and 812500 to 875000
         Summing of value range 750001 to 812500 is 48828156250
         Summing of value range 812501 to 875000 is 52734406250
Forking computation into two ranges: 875001 to 937500 and 937500 to 1000000
         Summing of value range 875001 to 937500 is 56640656250
         Summing of value range 937501 to 1000000 is 60546906250
Sum for range 1..1000000; computed sum = 500000500000, formula sum = 500000500000
```

Let's analyze how this program works. In this program, you want to compute the sum of the values in the range 1..1,000,000. For the sake of simplicity, you decide to use ten threads to execute the tasks. The class `RecursiveSumOfN` extends `RecursiveTask<Long>`. In `RecursiveTask<Long>`, you use `<Long>` because the sum of numbers in each sub-range is a `Long` value. In addition, you chose `RecursiveTask<Long>` instead of plain `RecursiveAction` because each subtask returns a value. If the subtask does not return a value, you can use `RecursiveAction` instead.

In the `compute()` method, you decide whether to compute the sum for the range or subdivide the task further using following condition:

```
(to - from) <= N/NUM_THREADS)
```

You use this "threshold" value in this computation. In other words, if the range of values is within the threshold that can be handled by a task, then you perform the computation; otherwise you recursively divide the task into two parts. You use a simple `for` loop to find the sum of the values in that range. In the other case, you divide the range similarly to how you divide the range in a binary search algorithm: for the range `from .. to`, you find the mid-point and create two sub-ranges `from .. mid` and `mid + 1 .. to`. Once you call `fork()`, you wait for the first task to complete the computation of the sum and spawn another task for the second half of the computation.

In the `main()` method, you create a `ForkJoinPool` with number of threads given by `NUM_THREADS`. You submit the task to the fork/join pool and get the computed sum for 1..1,000,000. Now you also calculate the sum using the formula to sum N continuous numbers.

From the output of the program, you can observe how the task got subdivided into subtasks. You can also verify from the output that the computed sum and sum computed from the formula are the same, indicating that your division of tasks for summing the sub-ranges is correct.

In this program, you arbitrarily assumed the number of threads to use was ten threads. This was to simplify the logic of this program. A better approach to decide the threshold value is to divide the data size length by the number of available processors. In other words,

```
 threshold value = (data length size) / (number of available processors);
```

How do you programmatically get the number of available processors? For that you can use the method `Runtime.getRuntime().availableProcessors()`).

In Listing 14-17, you used `RecursiveTask`; however, if a task is not returning a value, then you should use `RecursiveAction`. Let's implement a search program using `RecursiveAction`. Assume that you have a big array (say of 10,000 items) and you want to search a key item. You can use the Fork/Join framework to split the task into several subtasks and execute them in parallel. Listing 14-18 contains the program implementing the solution.

Listing 14-18. SearchUsingForkJoin.java

```
import java.util.concurrent.*;

//This class illustrates how we can search a key within N numbers using fork/join framework
// (using RecursiveAction).
//The range of numbers are divided into half until the range can be handled by a thread.
class SearchUsingForkJoin {
        private static int N = 10000;
        private static final int NUM_THREADS = 10; // number of threads to create for
                                                   // distributing the effort
        private static int searchKey= 100;
        private static int[] arrayToSearch;

        // This is the recursive implementation of the algorithm;
        // inherit from RecursiveAction
        static class SearchTask extends RecursiveAction {
                private static final long serialVersionUID = 1L;
                int from, to;
                // from and to are range of values to search
                public SearchTask(int from, int to) {
                         this.from = from;
                         this.to = to;
                }

                public void compute() {
                        //If the range is smaller enough to be handled by a thread,
                        //we search in the range
                        if( (to - from) <= N/NUM_THREADS) {
                               // add in range 'from' .. 'to' inclusive of the value 'to'
                               for(int i = from; i <= to; i++) {
                                       if(arrayToSearch[i] == searchKey)
                                               System.out.println("Search key found at index:" +i);
                               }
                        }
                        else {
                               // no, the range is big for a thread to handle,
                               // so fork the computation
                               // we find the mid-point value in the range from..to
                               int mid = (from + to)/2;
                               System.out.printf("Forking computation into two ranges: " +
"%d to %d and %d to %d %n", from, mid, mid, to);
                               //invoke all the subtasks
                               invokeAll(new SearchTask(from, mid),new SearchTask(mid + 1, to));
                        }
                }
        }
```

```
        public static void main(String []args) {
                //intantiate the array to be searched
                arrayToSearch = new int[N];
                //fill the array with random numbers
                for(int i=0; i<N; i++){
                        arrayToSearch[i] = ThreadLocalRandom.current().nextInt(0,1000);
                }
                // Create a fork-join pool that consists of NUM_THREADS
                ForkJoinPool pool = new ForkJoinPool(NUM_THREADS);
                // submit the computation task to the fork-join pool
                pool.invoke(new SearchTask(0, N-1));
        }
}
```

The program prints the following output (which might be different from run to run):

```
Forking computation into two ranges: 0 to 4999 and 4999 to 9999
Forking computation into two ranges: 0 to 2499 and 2499 to 4999
Forking computation into two ranges: 5000 to 7499 and 7499 to 9999
Forking computation into two ranges: 2500 to 3749 and 3749 to 4999
Forking computation into two ranges: 0 to 1249 and 1249 to 2499
Forking computation into two ranges: 2500 to 3124 and 3124 to 3749
Forking computation into two ranges: 7500 to 8749 and 8749 to 9999
Forking computation into two ranges: 5000 to 6249 and 6249 to 7499
Forking computation into two ranges: 8750 to 9374 and 9374 to 9999
Forking computation into two ranges: 5000 to 5624 and 5624 to 6249
Forking computation into two ranges: 7500 to 8124 and 8124 to 8749
Forking computation into two ranges: 3750 to 4374 and 4374 to 4999
Search key found at index:4736
Search key found at index:2591
Forking computation into two ranges: 1250 to 1874 and 1874 to 2499
Search key found at index:1315
Forking computation into two ranges: 0 to 624 and 624 to 1249
Search key found at index:445
Search key found at index:9402
Search key found at index:9146
Forking computation into two ranges: 6250 to 6874 and 6874 to 7499
Search key found at index:6797
Search key found at index:7049
Search key found at index:862
```

The key difference between Listings 14-14 and 14-15 is that you used `RecursiveAction` in the latter instead of `RecursiveTask`. You made several changes to extend the task class from `RecursiveAction`. The first change is that the `compute()` method is not returning anything. Another change is that you used the `invokeAll()` method to submit the subtasks to execute. Another obvious change is that you carried out search in the `compute()` method instead of summation in earlier case. Apart from these changes, the program in Listing 14-17 works much like the program in Listing 14-18.

Points to Remember

Remember these points for your exam:

- It is possible to achieve what the Fork/Join framework offers using basic concurrency constructs such as `start()` and `join()`. However, the Fork/Join framework abstracts many lower-level details and thus is easier to use. In addition, it is much more efficient to use the Fork/Join framework instead handling the threads at lower levels. Furthermore, using `ForkJoinPool` efficiently manages the threads and performs much better than conventional threads pools. For all these reasons, you are encouraged to use the Fork/Join framework.
- Each `worker` thread in the Fork/Join framework has a work queue, which is implemented using a `Deque`. Each time a new task (or subtask) is created, it is pushed to the head of its own queue. When a task completes a task and executes a join with another task that is not completed yet, it works smart. The thread pops a new task from the head of its queue and starts executing rather than sleeping (in order to wait for another task to complete). In fact, if the queue of a thread is empty, then the thread pops a task from the tail of the queue belonging to another thread. This is nothing but a work-stealing algorithm.
- It looks obvious to call `fork()` for both the subtasks (if you are splitting in two subtasks) and call `join()` two times. It is correct—but inefficient. Why? Well, basically you are creating more parallel tasks than are useful. In this case, the original thread will be waiting for the other two tasks to complete, which is inefficient considering task creation cost. That is why you call `fork()` once and call `compute()` for the second task.
- The placement of `fork()` and `join()` calls are very important. For instance, let's assume that you place the calls in following order:

  ```
  first.fork();
  resultFirst = first.join();
  resultSecond = second.compute();
  ```

  This usage is a serial execution of two tasks, since the second task starts executing only after the first is complete. Thus, it is less efficient even than its sequential version since this version also includes cost of the task creation. The take-away: watch your placement of fork/join calls.
- Performance is not always guaranteed while using the Fork/Join framework. One of the reasons we mentioned earlier is the placement of fork/join calls.

QUESTION TIME!

1. Consider the following program:

   ```
   import java.util.concurrent.atomic.*;

   class AtomicIntegerTest {
        static AtomicInteger ai = new AtomicInteger(10);
        public static void check() {
                assert (ai.intValue() % 2) == 0;
        }
        public static void increment() {
                ai.incrementAndGet();
        }
   ```

```
    public static void decrement() {
            ai.getAndDecrement();
    }
    public static void compare() {
            ai.compareAndSet(10, 11);
    }
    public static void main(String []args) {
            increment();
            decrement();
            compare();
            check();
            System.out.println(ai);
    }
}
```

The program is invoked as follows:

```
java -ea AtomicIntegerTest
```

What is the expected output of this program?

A. It prints 11.

B. It prints 10.

C. It prints 9.

D. It crashes throwing an `AssertionError`.

Answer:

D. It crashes throwing an `AssertionError`.

(The initial value of `AtomicInteger` is 10. Its value is incremented by 1 after calling `incrementAndGet()`. After that, its value is decremented by 1 after calling `getAndDecrement()`. The method `compareAndSet`(10, 11) checks if the current value is 10, and if so sets the atomic integer variable to value 11. Since the assert statement checks if the atomic integer value `% 2` is zero (that is, checks if it is an even number), the assert fails and the program results in an `AssertionError`.)

2. Which one of the following options correctly makes use of `Callable` that will compile without any errors?

A.
```
import java.util.concurrent.Callable;

class CallableTask implements Callable {
        public int call() {
                System.out.println("In Callable.call()");
                return 0;
        }
}
```

B.
```
import java.util.concurrent.Callable;

class CallableTask extends Callable {
        public Integer call() {
                System.out.println("In Callable.call()");
                return 0;
        }
}
```

C.
```
import java.util.concurrent.Callable;

class CallableTask implements Callable<Integer> {
        public Integer call() {
                System.out.println("In Callable.call()");
                return 0;
        }
}
```

D.
```
import java.util.concurrent.Callable;

class CallableTask implements Callable<Integer> {
        public void call(Integer i) {
                System.out.println("In Callable.call(i)");
        }
}
```

Answer:

C.
```
import java.util.concurrent.Callable;

class CallableTask implements Callable<Integer> {
        public Integer call() {
                System.out.println("In Callable.call()");
                return 0;
        }
}
```

(The `Callable` interface is defined as follows:

```
public interface Callable<V> {
    V call() throws Exception;
}
```

In option A), the `call()` method has the return type `int`, which is incompatible with the return type expected for overriding the `call` method and so will not compile.

In option B), the `extends` keyword is used, which will result in a compiler (since `Callable` is an interface, the `implements` keyword should be used).

Option C) correctly defines the `Callable` interface providing the type parameter `<Integer>`. The same type parameter `Integer` is also used in the return type of the `call()` method that takes no arguments, so it will compile without errors.

In option D), the return type of `call()` is `void` and the `call()` method also takes a parameter of type `Integer`. Hence, the method declared in the interface `Integer call()` remains unimplemented in the `CallableTask` class and so the program will not compile.)

3. Which one of the following methods return a `Future` object?

   A. The overloaded `replace()` methods declared in the `ConcurrentMap` interface

   B. The `newThread()` method declared in the `ThreadFactory` interface

   C. The overloaded `submit()` methods declared in the `ExecutorService` interface

   D. The `call()` method declared in the `Callable` interface

   Answer:

   C. The overloaded `submit()` methods declared in `ExecutorService` interface

   Option A) The overloaded `replace()` methods declared in the `ConcurrentMap` interface remove an element from the map and return the success status (a Boolean value) or the removed value.

   Option B) The `newThread()` is the only method declared in the `ThreadFactory` interface and it returns a `Thread` object as the return value.

   Option C) The `ExecutorService` interface has overloaded `submit()` method that takes a task for execution and returns a `Future` representing the pending results of the task.

   Option D) The `call()` method declared in `Callable` interface returns the result of the task it executed.)

4. You're writing an application that generates random numbers in the range 0 to 100. You want to create these random numbers for use in multiple threads as well as in `ForkJoinTasks`. Which one of the following options will you use for less contention (i.e., efficient solution)?

   A. `int randomInt = ThreadSafeRandom.current().nextInt(0, 100);`

   B. `int randomInt = ThreadLocalRandom.current().nextInt(0, 101);`

   C. `int randomInt = new Random(seedInt).nextInt(101);`

   D. `int randomInt = new Random().nextInt() % 100;`

   Answer:

   B. `int randomInt = ThreadLocalRandom.current().nextInt(0, 101);`

   (`ThreadLocalRandom` is a random number generator that is specific to a thread. From API documentation of this class: "Use of the `ThreadLocalRandom` rather than shared Random objects in concurrent programs will typically encounter much less overhead and contention."

   The method "`int nextInt(int least, int bound)`" in the `ThreadLocalRandom` class returns a pseudo-random number that is uniformly distributed between the given least value and the bound value. Note that the value in parameter `least` is inclusive of that value and the `bound` value is exclusive. So, the call `nextInt(0, 101)` returns pseudo-random integers in the range 0 to 100.)

5. In your application, there is a producer component that keeps adding new items to a fixed-size queue; the consumer component fetches items from that queue. If the queue is full, the producer has to wait for items to be fetched; if the queue is empty, the consumer has to wait for items to be added.

   Which one of the following utilities is suitable for synchronizing the common queue for concurrent use by a producer and consumer?

   A. `RecursiveAction`

   B. `ForkJoinPool`

   C. `Future`

   D. `Semaphore`

   E. `TimeUnit`

   Answer:

   D. `Semaphore`

   (The question is a classic producer–consumer problem that can be solved by using semaphores. The objects of the synchronizer class `java.util.concurrent.Semaphore` can be used to guard the common queue so that the producer and consumer can synchronize their access to the queue. Of the given options, semaphore is the only *synchronizer*; other options are unrelated to providing synchronized access to a queue.

   Option A) `RecursiveAction` supports recursive `ForkJoinTask`, and option B) `ForkJoinPool` provides help in running a `ForkJoinTask` in the context of the Fork/Join framework. Option C) `Future` represents the result of an asynchronous computation whose result will be "available in the future once the computation is complete." Option E) `TimeUnit` is an enumeration that provides support for different time units such as milliseconds, seconds, and days.)

Summary

Using java.util.concurrent Collections

- A semaphore controls access to shared resources. A semaphore maintains a counter to specify number of resources that the semaphore controls.
- `CountDownLatch` allows one or more threads to wait for a countdown to complete.
- The `Exchanger` class is meant for exchanging data between two threads. This class is useful when two threads need to synchronize between each other and continuously exchange data.
- `CyclicBarrier` helps provide a synchronization point where threads may need to wait at a predefined execution point until all other threads reach that point.
- `Phaser` is a useful feature when few independent threads have to work in phases to complete a task.

Applying Atomic Variables and Locks

- Java provides an efficient alternative in the form of atomic variables where one needs to acquire and release a lock just to carry out primitive operations on variables.
- A lock ensures that only one thread accesses a shared resource at a time.
- A `Condition` supports thread notification mechanism. When a certain condition is not satisfied, a thread can wait for another thread to satisfy that condition; that other thread could notify once the condition is met.

Using Executors and ThreadPools

- The `Executors` hierarchy abstracts the lower-level details of multi-threaded programming and offers high-level user-friendly concurrency constructs.
- The `Callable` interface represents a task that needs to be completed by a thread. Once the task completes, the `call`() method of a `Callable` implementation returns a value.
- A thread pool is a collection of threads that can execute tasks.
- `Future` represents objects that contain a value that is returned by a thread in the future.
- `ThreadFactory` is an interface that is meant for creating threads instead of explicitly creating threads by calling a new `Thread`().

Using the Parallel Fork/Join Framework

- The Fork/Join framework is a portable means of executing a program with decent parallelism.
- The framework is an implementation of the `ExecutorService` interface and provides an easy-to-use concurrent platform in order to exploit multiple processors.
- This framework is very useful for modeling divide-and-conquer problems.
- The Fork/Join framework uses the work-stealing algorithm: when a worker thread completes its work and is free, it takes (or "steals") work from other threads that are still busy doing some work.
- The work-stealing technique results in decent load balancing thread management with minimal synchronization cost.
- `ForkJoinPool` is the most important class in the Fork/Join framework. It is a thread pool for running fork/join tasks—it executes an instance of `ForkJoinTask`. It executes tasks and manages their lifecycles.
- `ForkJoinTask<V>` is a lightweight thread-like entity representing a task that defines methods such as `fork`() and `join`().