

**CHAPTER 13**

**Threads**



Create and use the Thread class and the Runnable interface

Manage and control thread lifecycle

**EXAM TOPICS**

Synchronize thread access to shared data

Identify potential threading problems

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(),*** | Overloaded | The current thread invoking join on another thread waits |
| ***void join(long),*** | instance methods | until the other thread dies. You can optionally give the |
| ***void join(long, int)*** |  | 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)*** | Overloaded static | Makes the current thread sleep for given milliseconds |
| ***void sleep(long, int)*** | methods | (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(),*** | Overloaded instance | The current thread should have acquired a lock on this |  |
| ***void wait(long),*** | methods | object before calling any of the wait methods. |  |
| ***void wait(long, int)*** |  | 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.

***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 staticmethod ***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 acouple 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.Main thread starts.

Thread myThread = new MyThread1();

|  |  |  |
| --- | --- | --- |
|  | myThread.start(); |  |
|  | JVM spawns a new thread and invokes the |  |
| Main thread continues | run() method. |  |
|  |  |
| execution | System.out.printIn("In run method; |  |
|  | thread name is: " + getName()); |  |

System.out.printIn("In main method;thread name is: " +

Thread.currentThread().getName());

***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 implementthe ***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 twocompiler 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 differencebetween 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 morethread 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 cannotdetermine), 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();***

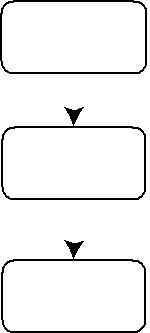
*The thread state is: TERMINATED*

***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.



new

Thread got started



runnable

Thread completed its task

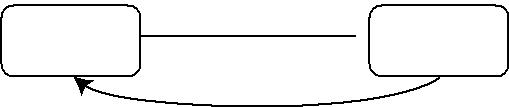


terminated

***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.



OS dispatches thread

ready running



timeout

***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:

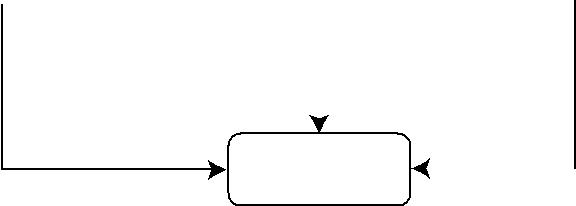
***3*** ***3 5 6 3 4 7 8 9***

***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.

thread t2 tries Counter.count++



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| thread t1 tries |  |  | thread t3 tries |  |
| Counter.count++ |  |  |  |
|  | Counter.count++ | |  |
|  |  |  |
|  |  |  |  |  |
|  | Counter.count | |  |  |
|  |  |  |

***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 alock 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 adeadlock (Figure 13-5).

t1 acquires lock on Balls.class.

|  |  |  |  |
| --- | --- | --- | --- |
| Balls.class | t2 waiting to lock |  |  |
|  |  |  |
|  | on Balls.class. |  |  |
| t1 waiting to lock |  | t2 acquired lock |  |
|  |  |  |
| on Runs.class. | Runs.class | on Runs.class. |  |

***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*** threadthat 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).

* + ***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.

* + ***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 programmaticallycontrol 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.

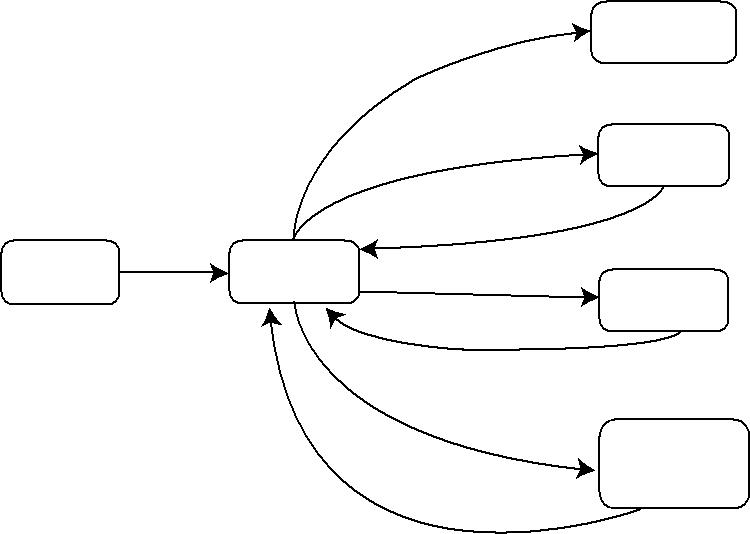
The ***DiceGame*** class does something very simple. It has the ***main()*** method and you create the ***Jane*** and ***Joe*** player objects. You set one of them to start the game. You call the ***start()*** methods for these two player threads to start playing.

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.



terminated

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | dies |  |  |
|  |  |  | thread |  |  |
|  |  |  | waiting for |  |  |
|  |  |  | acquiring lock | blocked |  |
|  |  |  |  |  |
|  | start() |  | lock acquired |  |  |
|  |  |  |  |  |
| new | called | runnable | join() or wait()called |  |  |
|  |  |  |
|  |  |  |  | waiting |  |
|  |  |  | notify()/notifyAll() |  |  |
|  |  |  | called |  |  |

sleep() or join() or

wait() with timeout

timed\_waiting

timeout completed

***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 forthe ***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 needto 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.*** 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 bythrowing 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.