[CHAPTER] 11

Threads

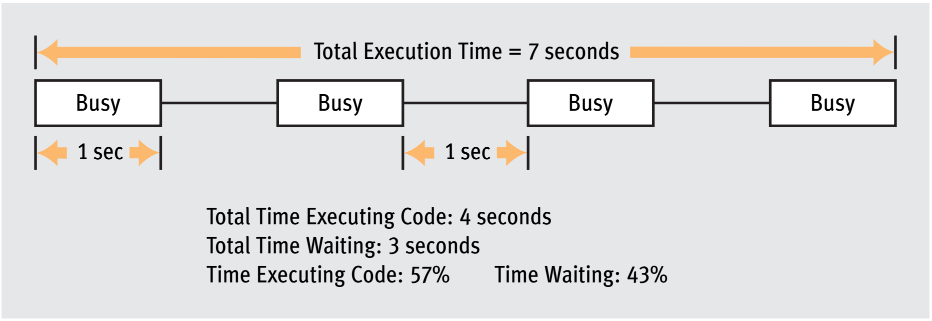
11.1 Introduction

People can quite easily do more than one thing at a time. Right now you are reading this text, and even though it is captivating, your brain is doing other work. While your eyes are scanning these words, your brain is simultaneously instructing your heart to beat and your lungs to fill with air. The ability to perform several tasks at the same time is called **multitasking**, and is a vital part of modern life. Many people must balance three or four tasks several times each day.

Modern computing systems also multitask; it is a common and powerful design technique. A chat program, for example, not only needs to communicate with the user, it must monitor the network for incoming connections and detect when new users have logged on to the system. A single task could accomplish all this, but the ability to implement these functions as multiple independent tasks makes software development easier and the underlying implementation more efficient.

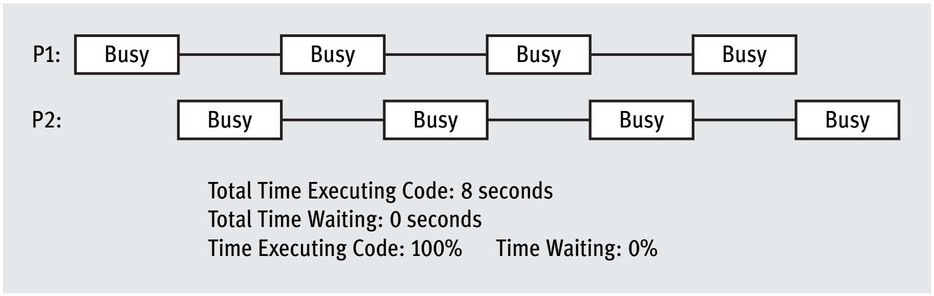
At first glance, you might think that restricting a computer to perform only one task at a time is an efficient use of computing resources, because the entire system is focused on one job. However, resources are actually being wasted because the processor may be idle for significant periods of time during the execution of a single program. A typical program does not spend 100 percent of its time computing. At various points during its execution, a program may need to wait for some event—for example, the program may wait for the user to enter data or for the next block of data to be read off the disk.

While a program waits for an event to take place, the processor is idle. If only one program is in memory, the processor can do nothing but wait. To make matters worse, a typical program (see Figure 11-1) often must wait several times during its lifetime for different events to take place. The program in Figure 11-1 takes a total of 7 seconds to execute; however, during that time, the program waits three times for a total of 3 seconds. Therefore, if this is the only program in the system, the processor is idle about 43 percent of the time. Clearly, to use a computing system to its fullest potential, the goal should be to use the processor 100 percent of the time for productive work.


Total Execution Time = 7 seconds
Busy
Busy
Busy
Busy
1 sec
1 sec
Total Time Executing Code: 4 seconds
Total Time Waiting: 3 seconds
Time Executing Code: 57%
Time Waiting: 43%

[FIGURE 11-1] Process execution profile

A primary goal of operating systems research is to improve efficiency by reducing the processor's idle time. Much of the early work in this field focused on ways to use the processor during periods of program inactivity. Researchers realized that if multiple programs were loaded into memory, it would be possible to interleave program execution, resulting in improved processor usage. Figure 11-2 illustrates how the execution of two programs, P1 and P2, can be interleaved so that one uses the processor when the other is inactive. In this case, the result is dramatic. Instead of being idle 43 percent of the time, the processor efficiency is nearly 100 percent.


P1:
Busy
Busy
Busy
Busy
P2:
Busy
Busy
Busy
Busy
Total Time Executing Code: 8 seconds
Total Time Waiting: 0 seconds
Time Executing Code: 100%
Time Waiting: 0%

[FIGURE 11-2] An improved process execution profile

Typically, the operating system decides how to interleave program execution so that the processor is never idle. In a **time-sharing**, or **multitasking**, operating system, the processor executes several programs by switching between them. This switching is done so quickly and frequently that system users are often unaware that the process is taking place. The **scheduler** is the part of the operating system that determines which program to run, when the current program should stop running, and when a new program should start. A primary goal of the scheduler is to increase the utilization of the processor.

A program that you can load into memory and then execute is called a **process**. The scheduler attempts to increase processor utilization by executing processes so that the processor always has useful work to do. Although it sounds complicated, the basic operation of the scheduler is easy to understand. It maintains a list, or **ready queue**, of processes that are ready to execute. Another queue, called the **waiting queue**, contains processes that are not ready to execute because they are waiting for an event to occur. The scheduler selects a process in the ready queue and arranges for it to be executed by the processor. The running process executes until it either terminates, is preempted, or it blocks (in other words, it has to stop and wait for an event to take place). When the scheduler detects that the current process is waiting, it moves the process to the waiting queue and selects a process in the ready queue for execution.

Not only does scheduling improve the utilization of computing systems, it allows programmers to use concurrency in their programs. **Concurrent programming** is a technique in which a program is divided into several independent tasks that are executed concurrently by the scheduler—that is, interleaved in time. Concurrency creates the illusion that more than one thing is happening at the same time. A computer with a single processor can obviously execute only one instruction and thus execute only one process at any instant; however, it can switch between processes very quickly and create the illusion that multiple processes are running simultaneously.

Although concurrent programming can help programmers develop sophisticated software, it also increases system overhead. Part of the problem is that while the scheduler is switching processes, no useful work is being done. During a **context switch**, when the scheduler removes the current process and brings in a new one, all of the information associated with the running process must be saved, and the state of the new process must be restored. In a modern operating system, a considerable amount of information is associated with the state of a process, which increases the amount of time required to perform a context switch. The processor does work during a context switch, but it is not useful work. During a context switch, only the scheduler is using the processor, so no user process can run. Context switches occur so frequently that even the slightest amount of wasted time can drastically affect overall system efficiency.

If the amount of information saved and restored during a context switch is reduced, the amount of time required to perform the switch is reduced as well. This idea led developers to create the concept of a **thread**, also called a **lightweight process**. A thread is a single flow of execution through a program. The amount of state information associated with a thread is significantly less than that of a process, so you can save and restore the state of a thread quickly. This leads to reduced context switching time, which increases overall system efficiency. To reduce the amount of information associated with a process, threads must often share resources. For example, all the threads in a Java program use the same memory space and system resources.

Prior to Java, scheduling was performed by the operating system. If you developed software that would run on a multitasking platform, you could use concurrency in your programs. Because most conventional operating systems support multitasking, this was not a big problem. However, software is now developed for more diverse computing environments, including cell phones, personal data assistants (PDAs), and even washing machines. Operating systems that manage these devices may not support concurrent programming.

Not only can your Java software run on any platform that offers a Java Virtual Machine (JVM), the software can also use concurrency regardless of the capabilities of the computing platform. The JVM itself provides the support required for scheduling threads in a Java program, rather than relying on the operating system's underlying computing services. As a result, Java programmers can use concurrency in programs whenever they need it, whether the program is running on a supercomputer, a PDA, or a washing machine.

Concurrency has become a standard tool for developing modern software. With few exceptions, the programs you run on a regular basis use concurrency. For example, browsers use separate threads to load the images on a Web page. Word processors simultaneously perform spelling checks in a separate thread as you type your document. Given the widespread use of concurrency in modern software systems, all programmers must be able to work with and understand programs that use multiple threads.

This chapter discusses how to use threads in a Java program. It starts by explaining how to create a thread and specify the code it executes. It then looks at the inner workings of the JVM scheduler to help you understand how threads interact with each other. The chapter concludes by examining the tools Java provides to synchronize threads so that the actions of one thread do not inadvertently affect the actions of another.

NOW THAT IS FAST

This chapter examines concurrency in programs and how it can simplify the design and implementation of software. Computers that contain more than one processor, such as dual-core machines, allow you to execute concurrent parts of a program in parallel, potentially speeding up the program. Think about performing a task such as washing a car. By yourself, it might take an hour to wash, dry, and wax the car. However, if three of your friends helped, you might be able to finish the task in less than 20 minutes. This is the basic idea behind parallel computing—a parallel computer with multiple processors can execute concurrent programs in less time.

Parallel computers are commonly used to solve computationally difficult problems. For example, a 10-day weather forecast is usually developed using computational models on a parallel computer. The first weather models were developed in the United Kingdom, and took a computer more than 24 hours to compute a one-day weather forecast. A 10-day weather forecast requires considerably more computing power, which is why many of these models are run on parallel computers.

The 500 most powerful computer systems in the world are listed at *www.top500.org*. At the time this book was published, the BlueGene/L system held top honors. The system contains more than 130,000 processors and can perform calculations at an astonishing rate of more than 280 teraflops (a teraflop is one trillion calculations per second). To put the speed of this machine into perspective, it takes 32,000 years for a trillion seconds to tick away. For a computer that executes a program at one teraflop, calculations that normally take an hour will only require 3.6 seconds; problems that normally require 100 days to solve will only require a few hours. Believe it or not, scientists are now talking about building machines that compute in the petaflop range. One petaflop equals 1000 teraflops, which is 1000 times faster than a teraflop.

11.2 Threads

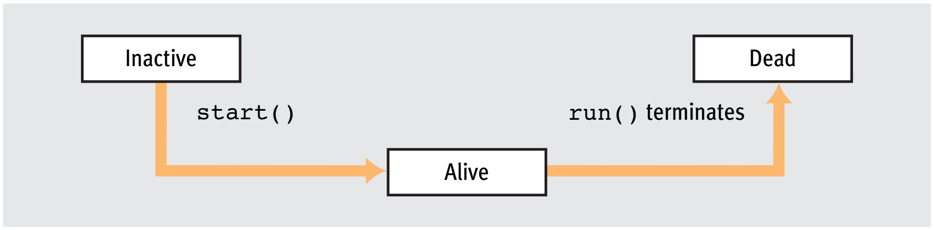
11.2.1 Creating Threads

A thread in a Java program is represented by an instance of the `Thread` class. The `Thread` class provides methods to create, access, modify, control, and synchronize a thread. An instance of the `Thread` class contains the information required to manage the execution of a thread. The key methods of the `Thread` class are listed in Table 11-1.

METHOD	DESCRIPTION
`static Thread currentThread()`	Returns a reference to the thread that is currently executing
`String getName()`	Returns the name of the thread
`int getPriority()`	Returns the priority of the thread
`void interrupt()`	Interrupts the thread
`Boolean isAlive()`	Returns true if this thread has been started and has not yet died
`void join()`	Returns when the thread has died
`void run()`	If this thread was constructed using a separate `Runnable` run object, then the object's run method is called; otherwise, the method does nothing and returns
`void setDaemon( )`	Marks the thread as a daemon thread
`void setName()`	Changes the name of the thread
`void setPriority()`	Changes the priority of the thread
`static void sleep()`	Causes the current thread to temporarily cease execution for the specified number of milliseconds
`void start()`	Causes this thread to be scheduled; when the thread is activated for the first time, the JVM invokes the `run()` method of the thread
`static void yield()`	Causes the current thread object to temporarily pause and allow other threads to execute

[TABLE 11-1] Class `Thread` methods

To create a thread in a Java program, you instantiate an object from the `Thread` class. The thread represented by this object remains inactive until its `start()` method is invoked. The `start()` method informs the scheduler that the thread is ready to run and that it should be put in the ready queue and scheduled for execution. Invoking the `start()` method does not cause the thread to begin executing; it only informs the scheduler that it is ready to run. The scheduler may be managing many other threads that are either waiting for some event or using the processor. In this situation, the newly activated thread might have to wait before it can start executing. A bit of time will probably pass between the moment the `start()` method is invoked and the moment the new thread actually begins execution. After `start()` is called, the new thread is in the state labeled *Alive* in Figure 11-3.



[FIGURE 11-3] Stages in thread activation

A thread is considered alive if the scheduler is aware of it and is managing its execution. However, remember that even though the scheduler manages the execution of a thread, it does not mean the thread is actually running on the processor. The thread may be ready to run but waiting for the processor, waiting for some event to occur, or in the process of terminating. The method `isAlive()`, defined in the `Thread` class, returns true if a thread has been started but has not yet died. If the `isAlive()` method returns true, you know that the scheduler is managing the thread, but you can't say for certain what it is actually doing.

The `run()` method contains the code that the thread executes. The scheduler invokes `run()` when the thread executes for the first time. You can invoke the `run()` method of a `Thread` object just as you invoke a method on any other object, and it executes just like any other method we have seen. However, the `run()` method is special because it is invoked automatically by the JVM when the thread starts to execute code. The role the `run()` method plays for a thread is similar to the role the `main()` method plays in a Java program. The `main()` method is invoked by the JVM when the program starts, just as `run()` is invoked by the JVM when the thread obtains control of the processor for the first time.

A thread remains alive until the `run()` method terminates. This method terminates like any other method in Java, by returning or throwing an exception. In either case, once the `run()` method returns, the thread is no longer alive and enters the state labeled *Dead* in Figure 11-3. When a thread dies, the `Thread` object that represents it is still valid and can access the state of the thread; however, it cannot be restarted or rerun.

A programmer must be able to specify the code that a thread executes when it is started. You can define the `run()` method for a thread in two ways. The first is to use inheritance to override the default behavior of the `run()` method defined in the `Thread` class. You do this by writing a class that extends `Thread` and provides its own `run()` method. The program in Figure 11-4, which creates and starts a thread that prints the message "I am a Thread," illustrates this technique.

```
/**
 * Create an additional thread in a Java program by extending
 * the Thread class.
 */
public class ExtendThread extends Thread {
    /**
     * This code is executed after the thread starts.
     * This method prints the message "I am a Thread"
     * on standard output and then terminates.
     */
    public void run() {
      System.out.println( "I am a Thread" );
    }

    /**
     * Create an instance of this class and schedule it for
     * execution.
     *
     * @param args command line arguments (ignored)
     */
    public static void main( String args[] ) {
      // Create the thread
      Thread theThread = new ExtendThread();

      // Start the thread. Note that this method returns
      // immediately. The new thread may or may not
      // be executing code when it returns.
      theThread.start();
    }

} // ExtendThread
```

[FIGURE 11-4] Specifying `Thread` code using inheritance

The second way to specify the code executed by a thread is to create a new class that implements the `Runnable` interface. This interface specifies exactly one method, `run()`, which identifies the code that is executed when a thread becomes active. When using the `Runnable` interface, you must create two objects: an object that implements `Runnable` and a `Thread` object that represents the thread. When the `Thread` object is created, it is passed a reference to the `Runnable` object as a parameter to its constructor. When `start()` is invoked on the `Thread` object, it in turn invokes the `run()` method in the `Runnable` object. The code in Figure 11-5 works just like the code in Figure 11-4, but it uses the `Runnable` interface to specify which code is executed by the thread.

```java
/**
 * Illustrate how to create an additional thread in a Java
 * program by implementing the Runnable interface.
 */
public class ImplementRunnable implements Runnable {
    /**
     * This code is executed after the thread starts.
     * This method prints the message "I am a Thread"
     * on standard output and then terminates.
     */
    public void run() {
      System.out.println( "I am a Thread" );
    }

    /**
     * Create an instance of this class and a thread to run it.
     * Then start the thread so that the run() method executes.
     *
     * @param args command line arguments (ignored)
     */
    public static void main( String args[] ) {
      // Create the object that contains the run() method
      Runnable runnableObject = new ImplementRunnable();

      // Create the thread and tell it about the runnable object.
      Thread theThread = new Thread( runnableObject );

      // Start the thread.
      theThread.start();
    }

} // ImplementRunnable
```

[FIGURE 11-5] Specifying thread code by implementing the `Runnable` interface

Only one hard-and-fast rule exists for how you specify the code that a thread executes. Java does not support multiple inheritance; thus, if the class that contains the `run()` method for the thread already extends a class, it cannot extend `Thread`. In this situation, the class must implement the `Runnable` interface to execute the `run()` method in a separate thread. Many programmers prefer to specify the code a thread executes using the `Runnable` interface because it separates the code the thread executes from the state that must be maintained to manage the thread. The `Thread` object contains all of the state information required to manage the thread, and the `Runnable` object contains the code and application state that is executed by the thread.

You may not have realized it, but you have used threads in every Java program you have executed. When the JVM starts executing a Java program, it creates a thread that invokes the `main()` method of the class specified on the command line that started the program. Thus, the programs in Figures 11-4 and 11-5 actually have at least two active threads: the thread running `main()` and the thread printing "I am a Thread". A Java program might have as many as a dozen threads active at any one time. So what happens in Figure 11-4 if the thread running the `main()` method finishes before the `run()` method? In other words, how does the JVM know when it is safe to terminate the entire program? Clearly, if multiple threads are still active in a program, you do not necessarily want the JVM to terminate when the `main()` method has completed.

The JVM terminates the execution of a program when one of two conditions occurs. Under normal circumstances, the JVM terminates when all user threads have terminated. Notice that we said *all* user threads. If the `run()` method in one thread returns or a thread throws an exception and terminates, the JVM does not terminate unless the thread was the last user thread being executed. The same applies to the `main()` method. The JVM does not terminate when `main()` returns unless no other user threads are active in the program.

Threads can also be marked as a nonuser or daemon thread. A **daemon thread** is often used to provide a service in a Java program. For example, in a chat program, you may want to have a thread scan the network to determine who is using the service. This way, you can be chatting with Laura and be notified when Linus or Tom logs on to the system. Remember that the JVM terminates when all the user threads have terminated, not simply when all the threads have terminated. In other words, the JVM does not wait for daemon threads to finish before it terminates. By default, threads are marked as user threads. You can mark a thread as a daemon thread when the thread is constructed or by invoking the `setDaemon()` method on the thread.

The JVM also terminates when the `exit()` method of class `Runtime` is called (for example, `System.exit(0)`). In this case, the JVM terminates immediately, regardless of the number of active user threads in the program. Essentially, the `exit()` method aborts the execution of the JVM. Because the consequences of executing the `exit()` method are so drastic, you should only use it in special situations—typically, when an unrecoverable event has occurred in the program and all hope of continuation is lost. It is also appropriate to invoke the `exit()` method during the processing of command-line arguments in the `main()` method. If these arguments are incorrect and the program cannot even start, you might need to terminate it by invoking `exit()`.

Other attributes associated with `Thread` objects are useful when working with threads. One such attribute is a string that you can use to associate a name with the thread. This can be useful when working with programs that use multiple threads. The name of a thread can be specified as an argument to the constructor when creating the thread or by invoking the `setName()` method on the `Thread` object after it has been created. Once a name is associated with a thread, that name appears in diagnostic messages generated during run time and can be accessed by invoking the `getName()` method on the object. User-supplied thread names are optional. If you do not supply a name, a unique name is generated for the thread when it is created. The program in Figure 11-6 creates five different threads, each with a unique name. When the threads execute, they print their name to standard output and terminate.

```
/**
 * Create five threads, each with a different name. Each thread
 * prints its name on standard output and terminates.
 */
public class MultipleThreads extends Thread {
    // The number of threads to create
    public final static int NUM_THREADS = 5;

    /**
     * Print the name of the thread on standard output.
     */
    public void run() {
      System.out.println( getName() );
    }

    /**
     * Create NUM_THREADS threads, start each thread, and terminate.
     * The name of each thread is of the form "Thread #N", where
     * N is replaced by a number from 0 to NUM_THREADS - 1.
     *
     * @param args command line arguments (ignored)
     */
    public static void main( String args[] ) {
      Thread newThread = null;  // Reference to new thread
      String name = "Thread #"; // Prefix for all thread names

      // Create the threads
      for ( int i = 0; i < NUM_THREADS; i++ ) {
          // Create a thread
          newThread = new MultipleThreads();

          // Set the name
          newThread.setName( name + i );

          // Start the thread
          newThread.start();
      }
    }

} // MultipleThreads
```

[FIGURE 11-6] Example of a program with multiple threads

After examining Figure 11-6, it is reasonable to ask what output the program generates. Surprisingly, we cannot specify the exact output because it depends on the order in which the threads are executed; the JVM determines this order at run time. In fact, the Java language specification only guarantees that statements within a single thread are executed in the order specified by the program, but says nothing about how statements in different threads are interleaved during the execution of a program. Figure 11-7 shows four different results generated by executing the program in Figure 11-6 four times.

```
Thread #0     Thread #1     Thread #4     Thread #4
Thread #1     Thread #3     Thread #3     Thread #3
Thread #2     Thread #0     Thread #1     Thread #2
Thread #3     Thread #2     Thread #2     Thread #1
Thread #4     Thread #4     Thread #0     Thread #0
```

[FIGURE 11-7] Possible output from the program in Figure 11-6

Using threads can be a little disconcerting because it means that the programs you write can be **nondeterministic**. When you wrote and analyzed programs that used only a single thread, you could trace the execution of the program and be certain of the output that would be generated for a given set of inputs. In a multithreaded program, you can trace the execution of each thread, and you can even determine the output that each thread generates. However, you cannot determine how the scheduler interleaves the statements within the threads. Multiple runs of the same program may produce different results each time, because of the decisions made by the scheduler each time the program executes.

With that said, multiple runs of the same program on the same machine and under the same conditions often produce the same output. You are most likely to see different outputs if you run the same program on different platforms (for example, UNIX, Windows, Macintosh) because their schedulers may use different scheduling algorithms. The crucial point to remember is that with multithreaded programs, you cannot make assumptions about the order in which the scheduler executes threads. If a particular interleaving of the threads is required for correct execution, you must program that coordination using Java's thread synchronization features, which is discussed in Section 11.3.

Threads allow you to write Java programs that appear to do more than one thing at a time. Threads cannot always perform multiple actions simultaneously because most modern computers still have only one processor to execute user programs. However, these single-processor machines can switch from one thread to another so quickly that several things seem to be happening at once. The scheduler within the JVM selects which thread to execute and decides when to switch to another thread. The next section discusses the life cycle of a thread and how the scheduler manages the execution of threads within a Java program.

11.2.2 Scheduling and Thread Priorities

Because most modern computers typically have only a single processor, a mechanism must be in place that allows multiple threads to share the processor. The scheduler, an integral part of the JVM, must allocate the processor so that each thread can complete its work as efficiently as possible. At first glance, it might seem that the best approach to this task is to run one thread at a time, one after another, until they have all terminated. Each thread gets the processor time it requires to complete its work. However, the processor is not fully used because the processor may be idle for long periods of time; the current thread must wait for an event to occur before it can continue.

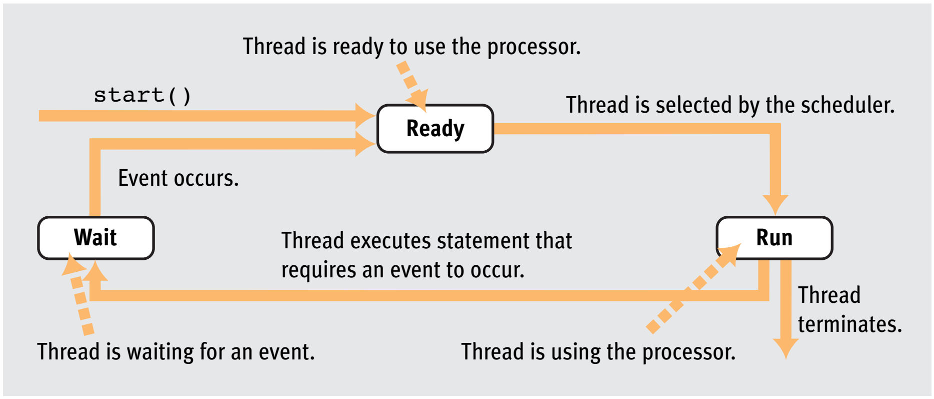
The way the JVM schedules the threads in a Java program is similar to how patients are scheduled in a doctor's office. Several patients (threads) require the services of a single doctor (the processor). In a typical doctor's office, patients are not treated one at a time, but are scheduled so that the doctor can see several patients at the same time. Patients are placed in different examination rooms; the doctor selects one of the rooms and treats the patient in that room. If the current patient requires tests, the doctor orders the tests and sees other patients until the tests are finished. When a patient's tests are finished, the doctor continues treating that patient. Even the waiting room is designed to improve the doctor's utilization. Having several patients in the waiting area almost guarantees that when the doctor is ready to see a patient, one will be ready. The scheduling system attempts to ensure that each patient is treated within a reasonable amount of time and that the doctor's time is used as efficiently as possible.

The typical thread follows a specific pattern, which is what led computer scientists to develop scheduling algorithms. Almost all threads are in one of three states while they are running. A thread is in the **ready state** when it is ready to use the processor but has not yet been given the processor. Once it is given the processor, it can start executing code immediately. A thread is in the **run state** if the processor is currently executing it. On a single processor machine, only one thread is in the run state. Multiple processor machines may have more than one running process (however, this text assumes that only one is running). Finally, a thread that is waiting for an event to occur is in the **wait state**. A thread remains in the wait state until the event it awaits has occurred.

Patients who are waiting for a doctor have the same states. Patients waiting for a doctor in examination rooms are in the ready state. The patient being examined is in the run state. Finally, patients waiting for test results or completed paperwork are in the wait state. The doctor is busy most of the time because at least one patient is almost always in the ready state. When the doctor finishes with one patient, another is ready to be examined. The staff is responsible for monitoring the state of the patients and ensuring that the doctor always has a patient to see.

Like the staff in the doctor's office, the JVM scheduler must track the state of the threads in a Java program and manage their execution so one is always available when the processor is ready to run a thread. The scheduler uses two queues to keep track of the threads it is managing. The **ready queue** contains all the threads that are in the ready state and are prepared to use the processor. The **wait queue** contains all the threads that are blocked, waiting for some event to occur. When the `start()` method of a thread is invoked, the thread is placed in the ready queue and competes with the other ready threads to gain access to the processor. This is why you cannot assume that a thread is executing code immediately after invoking its `start()` method.

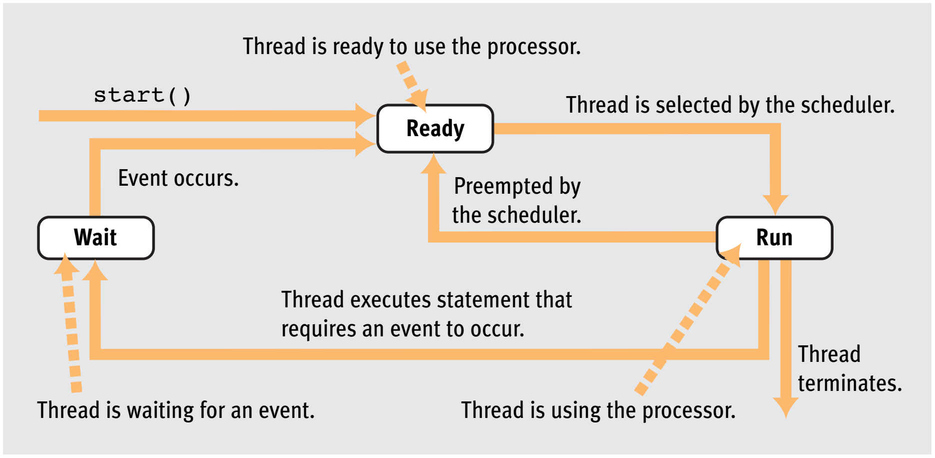
Because most threads run for short periods of time and then wait for an event to take place before they can continue, the scheduler can arrange things so that the processor almost always has useful work to do. If the scheduler is managing the execution of several threads, then when one of the threads blocks, the scheduler can move it to the wait queue and select the next thread from the ready queue to run. Things rarely work out as efficiently for the scheduler, as shown in Figure 11-2. How effectively the scheduler can use the processor depends on the computing requirements of the threads it is managing.


Thread is ready to use the processor.
start()
Ready
Thread is selected by the scheduler.
Event occurs.
Wait
Thread executes statement that requires an event to occur.
Run
Thread terminates.
Thread is waiting for an event.
Thread is using the processor.

[FIGURE 11-8] Thread states

The basic operations performed by a scheduler are illustrated in Figure 11-8. When the scheduler determines that the processor is idle, it selects a thread from the ready queue and prepares it for execution. The thread is then allowed to run until it terminates or has to wait for an event to occur before it can continue. The scheduler then either marks the thread as finished or moves the thread to the queue of waiting threads. The processor is again idle, and another thread is selected from the ready queue and allowed to run. This cycle repeats until all the threads have terminated, at which time the JVM can terminate. In addition to keeping the processor busy, the scheduler also monitors the threads in the wait queue. The scheduler moves a thread from the wait queue to the ready queue when the event the thread was awaiting has occurred.

The type of scheduler in Figure 11-8 is called a **nonpreemptive** scheduler because once a thread obtains control of the processor, it keeps control until it terminates, executes an instruction that causes the thread to wait, or yields the processor to another thread. Nonpreemptive schedulers are fairly easy to implement, but they allow a greedy thread to monopolize the use of the processor. A **preemptive** scheduler, on the other hand, interrupts the execution of a thread so that other threads may use the processor. Figure 11-9 illustrates the actions of a preemptive scheduler. The only difference between the two types of schedulers is the addition of a transition from the running state directly to the ready state. This occurs when the thread is interrupted by the scheduler.


Thread is ready to use the processor.
start()
Ready
Thread is selected by the scheduler.
Event occurs.
Preempted by the scheduler.
Wait
Run
Thread executes statement that requires an event to occur.
Thread terminates.
Thread is waiting for an event.
Thread is using the processor.

[FIGURE 11-9] Preemptive scheduling

In a preemptive scheduler, the running thread can be preempted for a variety of reasons. Many preemptive schedulers assign a maximum length of time, called a **time quantum**, that defines how long a thread can remain in the running state. If the thread does not terminate or execute an instruction that initiates a context switch before the time quantum is exceeded, the scheduler preempts the thread and moves it to the ready queue. Instead of keeping track of how long a thread has been executing, most Java schedulers count the number of instructions the current thread has executed. The thread is preempted if it attempts to execute more than the maximum number of instructions allowed by the scheduler. The use of preemption allows for a fairer allocation of processing time to all the ready threads.

Many schedulers also allow programmers to assign priorities to threads. The scheduler then attempts to ensure that the highest-priority thread in the system is always executing. If a thread enters the ready queue and has a priority above that of the currently running thread, the current thread is preempted and the higher-priority thread executes. Thread priority affects the way in which the scheduler places threads in the ready queue. As you learned in Chapter 6, a priority-based scheduler often uses a priority queue to hold the ready threads. The higher-priority threads are at the front of the queue and therefore always have a chance to obtain the processor before the lower-priority threads at the back of the queue. In the case of ties, the threads are selected based on the order in which they were placed in the queue. A major problem with priority-scheduling algorithms is the possibility of **indefinite blocking** or **starvation**. Priority scheduling can leave some low-priority threads waiting indefinitely for the processor.

An integer priority is part of the state associated with every instance of the `Thread` class. When a new thread is created, it has the same priority as the thread that created it. You can explicitly set and determine the priority of a thread by invoking its `setPriority()` and `getPriority()` methods, respectively. Both methods are described in Table 11-1. Thread priorities range between the values specified by the class constants `MIN_PRIORITY` and `MAX_PRIORITY` defined in the `Thread` class (the class constant `NORM_PRIORITY` defines the default priority for a thread). High priorities are represented by large integer values, and low priorities are represented by small integer values. In Java, the maximum priority is currently 10, the minimum priority is zero, and the default priority is five. When programming, you should always use the class constants to specify priorities because the actual values may change in future releases of the language.

Given that all threads have a priority either explicitly assigned or set by default, you would think that the Java thread scheduler would be required to use priorities when making scheduling decisions. You would be wrong! As it turns out, the Java language specification states:

> *When there is competition for processing resources, threads with higher priority are generally executed in preference to threads with lower priority. However, such preference does not guarantee that the highest-priority thread is always running.*

Thus, the scheduler may use a thread's priority to help determine which thread to run next, but it is not required when making scheduling decisions. This allows the scheduler to have the flexibility it requires to ensure that all threads, regardless of their priority, receive some minimally acceptable level of service. Thread priorities serve as "hints" to the scheduler to help it determine the best way to schedule the threads and to avoid starvation. For this reason, use priorities to provide the scheduler with the information it needs to schedule your threads in the most efficient and beneficial way possible. For example, the thread in a word processor that writes a backup copy of the current document to the disk should have a lower priority than the thread that is accepting and interpreting keystrokes. Although both tasks are important, the word processor should spend most of its time examining the input. Otherwise, users become frustrated with the noticeable time lag between key presses and the appearance of the corresponding characters on the screen.

Never assume that the scheduler executes the threads in a program in priority order. Some threads in a multithreaded Java program need to coordinate their actions with other threads for the entire program to operate correctly. For example, in a word-processing program, the thread that monitors keyboard activity should not add new characters to the current document if the thread that automatically backs up the document is saving it to disk. In this case, the keyboard thread should wait until the backup thread completes the write operation before continuing. Without taking scheduling into consideration, you might have written code similar to that in Figure 11-10, which causes a thread to pause for a short time.

```
public class BusyWait extends Thread {
    // Number of times to execute the loop
    private static final int DELAY = 100000;

    public void run() {
      // Code omitted

      // Delay execution of this thread for a few milliseconds
      // using an empty for loop
      for ( int i = 0; i < DELAY; i++ ) {
          // Do nothing
      }

      // Continue execution

      // Code omitted
    }

    public static void main( String args[] ) {
      // Create the thread and start the thread
      new BusyWait().start();
    }

} // BusyWait
```

[FIGURE 11-10] Noncooperative thread

Consider what happens when the code in Figure 11-10 is executed in a multithreaded environment. The for loop in the `run()` method contains no instructions that cause the thread to relinquish the processor, thus providing the scheduler with the opportunity to perform a context switch. In fact, if the execution of this code were being managed by a non-preemptive scheduler, no other threads could execute until the for loop terminated. Effectively, this program waits by keeping the processor busy doing useless work, and because nothing can be scheduled during the wait, this thread delays the execution of the entire program. Instead of monopolizing the processor, a better coding strategy would be to have the thread voluntarily relinquish control of the processor so that the scheduler could effectively use the processor during the wait period. In **cooperative multitasking**, a thread voluntarily yields control of the processor, allowing other threads to execute. The next section explains some of the methods defined in the `Thread` class that a programmer can use to write cooperative threads.

11.2.3 Cooperative Multitasking

When writing code for a multithreaded environment, make sure that the various threads do not interact in unexpected ways. Even though we cannot determine the exact order in which threads are executed by the scheduler, we must be able to determine if the program works correctly. Consider the program in Figure 11-11. The main thread, created and started by the JVM, starts a user thread that performs some complex computation; the details are not important to this discussion. The main thread waits for the computation to finish and then prints the results. The main thread uses the `isAlive()` method to determine when the scheduler is finished with the user thread.

```
/**
 * Use a user thread to perform some complicated, time-consuming
 * computation.
 */
public class WorkerThread extends Thread {
    private static int result = 0; // Result of computation

    public void run() {
      // Perform a complicated, time-consuming calculation
      // and store the answer in the variable result
      // Code omitted
    }

    /**
     * Create and start thread to perform computation. Wait until
     * the thread has finished before printing the result.
     *
     * @param args command-line arguments (ignored)
     */
    public static void main(String args[]) {
      // Create and start the thread
      Thread t = new WorkerThread();
      t.start();

      // Wait for thread to terminate
      while ( t.isAlive() ) {
          // Do nothing
      }
```

continued

```
      // Print result
      System.out.println( result );
    }

} // WorkerThread
```

[FIGURE 11-11] Example of a busy wait loop

The program in Figure 11-11 contains a busy wait loop. The busy loop in this program takes the form of a while loop that continues to execute until the user thread that was created by the program terminates (in other words, when `t.isAlive()` becomes false). Unfortunately, the scheduling problem that occurred in Figure 11-10 can also happen here. Once the processor starts the busy loop in `main()`, the user thread cannot gain control of the processor until the busy loop terminates. However, in this program, the situation is considerably worse. In Figure 11-10, you know that the wait loop eventually terminates, but it does not terminate in Figure 11-11 until the user thread is no longer active. The user thread remains active until its `run()` method returns. For the `run()` method to return, the user thread must obtain sufficient processor time to do its work. But if the busy loop is executing, it never terminates because it monopolizes all of the processor time and prevents the user thread from doing any work at all. The result is that we might never get our answer.

Understand that events must occur in a very specific order to produce this situation. For example, if the scheduler decided to make the main thread wait until the user thread terminated, the deadlock would be eliminated. However, even if a deadlock is only possible rather than certain, it still means that the program cannot reliably perform correctly at all times, which makes it useless. Imagine for a moment if a similar deadlock *might* occur in the flight control software of an aircraft. You would not want to fly if there was even a tiny possibility of a deadlock occurring in midflight. Restarting the flight control computer is not an option when the plane is cruising at 37,000 feet.

This problem occurred because the scheduler does not have enough detailed information about the threads to execute them in the right order and thus prevent deadlock. The scheduler cannot prevent deadlock on its own; the programmer must avoid writing code that may potentially deadlock the system. One way to accomplish this is to provide hints to the scheduler about how threads should be scheduled. In Figure 11-11, the deadlock could have been avoided if the while loop contained an instruction that forced a context switch to occur. During the context switch, the scheduler can give the processor to the user thread and allow it to do useful work. However, how can the running thread tell the scheduler to switch to another thread? Three methods defined in the Java `Thread` class—`sleep()`, `yield()`, and `join()`—can cause a context switch to take place.

The `sleep()` method, a static method of the `Thread` class, causes the current thread to cease execution for a user-specified time interval. The `sleep()` method takes a single integer parameter that specifies the amount of time to sleep in milliseconds (one-thousandths of a second). Figure 11-12 shows how you could use the `sleep()` method to prevent the deadlock situation that arises in Figure 11-11.

```java
/**
 * Use a user thread to perform some complicated, time-consuming
 * computation. The main thread waits for the worker thread
 * using the sleep() method.
 */
public class SleepingMain extends Thread {
    private static final int DELAY = 1000; // Milliseconds to sleep
    private static int result = 0;         // Result of computation

    public void run() {
      // Perform a complicated, time-consuming calculation
      // and store the answer in the variable result

      // Code omitted
    }

     /**
      * Create and start thread to perform computation. Wait until
      * the thread finishes before printing the result.
      *
      * @param args command-line arguments (ignored)
      */
    public static void main(String args[]) {
      // Create and start the thread
      Thread t = new SleepingMain();
      t.start();

       // Wait for thread to terminate
      while ( t.isAlive() ) {
          try {
            Thread.sleep( DELAY );
          } catch ( InterruptedException e ) {}
      }

      // Print result
      System.out.println( result );
    }

} // SleepingMain
```

[FIGURE 11-12] Using `sleep()` to delay

By invoking the `sleep()` method within the while loop, the running processor gives up, or yields, the processor each time it executes the loop. When the `sleep()`method executes, the scheduler places the current thread in the wait queue for the number of milliseconds specified by the argument (1000 in this case). The use of `sleep()` in Figure 11-12 eliminates the situation that arose in Figure 11-11. That situation cannot occur now because `main()`gives up the processor for at least 1 second whenever it determines that the user thread is still active. During this sleep period, the scheduler is free to run the user thread, which means that it should eventually terminate.

You can interrupt a sleeping thread by invoking the `interrupt()` method on the thread. If the thread is sleeping when it is interrupted, the `sleep()` method throws an `InterruptedException` that must be caught or declared, as discussed in Chapter 10. If the user thread in Figure 11-12 is interrupted, the `InterruptedException` is caught but ignored.

The `sleep()` method causes only the current thread, the one that invoked the method, to sleep. You cannot use `sleep()` to put another thread to sleep. The `sleep()` method is static, so the sleep behavior is associated with the `Thread` class and not with any of its instances. The parameter passed to `sleep()` specifies the duration of the sleep period, making it impossible to specify which thread should sleep.

Like any static method, you can invoke `sleep()` by specifying the class in which the method is defined or by using a reference to an instance of the class. When using an instance reference to invoke `sleep()`, you can easily become confused about which thread is put to sleep. Consider the program in Figure 11-13, which sleeps for a total of three seconds by invoking `sleep()` three times. In each statement, the `sleep()` method is invoked in a different way, yet each time the current thread is the one that sleeps. The static method `currentThread()` obtains a reference to the thread being executed by the processor.

```
/**
 * Demonstrate different ways to invoke sleep.
 */
public class SleepingMain {
    public static final int ONE_SECOND = 1000;

    public static void main( String args[] ) {
      Thread t = new Thread();     // Create a thread Thread
      me = Thread.currentThread(); // A reference to the thread
                                   // executing main()

      t.start();                   // Start the thread

      // Each of the following statements instructs the current
      // thread to sleep for one second
      try {
          Thread.sleep( ONE_SECOND );
          me.sleep( ONE_SECOND );
```

continued

```
            t.sleep( ONE_SECOND ); // This puts main to sleep—NOT t!!
        } catch ( InterruptedException e ) {}
    }

} // SleepingMain
```

[FIGURE 11-13] Invoking `sleep()`

The first time the `sleep()` method is invoked in Figure 11-13, the class name `Thread` specifies the class that contains the method we want to invoke. The second time, the reference to the current thread, obtained previously in the program, is used to suspend execution of the thread. Both approaches are valid ways to put the current thread to sleep. However, the third time `sleep()` is used, you might be confused. Remember that `sleep()` is a static method, so even though we seem to be invoking `t`'s `sleep()` method, the current thread is actually the one that is suspended. Remember that the `sleep()` method is static, so regardless of how we access it, we are referring to the same `sleep()` method, and no matter how `sleep()` is called, the current thread is always the one that is suspended.

You might be tempted to use `sleep()` to introduce delays that last for a specific time in your programs. For example, Figure 11-14 contains a crude clock program written in Java. The program consists of an infinite loop that uses `sleep()` to delay execution for one second and then prints the number of seconds that have elapsed since the program started. The clock provided by this program is far from accurate—if you execute this program on almost any machine, it starts to lose time almost immediately.

```
/**
 * Once every second, print the elapsed time since the program
 * was started.
 */
public class Clock {
    public static final int ONE_SECOND = 1000; // Millisecs in a second
    public static final int SECS_PER_MIN = 60; // Seconds in a minute
    public static final int MINS_PER_HR = 60;  // Minutes in an hour

    /**
     * The main program consists of an infinite loop that sleeps for
     * one second, updates the current time, and prints the result.
     *
     * @param args command-line arguments (ignored)
     */
    public static void main( String args[] ) {
      int secs = 0, mins = 0, hrs = 0;  // Elapsed time
```

continued

```
      while ( true ) {
          // Sleep for one second
          try {
            Thread.sleep( ONE_SECOND );
          }
          catch ( InterruptedException e ) {}

          // Update the time
          secs++;

          if ( secs >= SECS_PER_MIN ) {
            secs = 0; mins++;

            if ( mins >= MINS_PER_HR ) {
                mins = 0; hrs++;
            }
          }

          // Print the current time
          System.out.println( hrs + ":" + mins + ":" + secs );
      }
    }

} // Clock
```

[FIGURE 11-14] Computation of elapsed time using `sleep()`

For the program in Figure 11-14 to keep the time accurately, the amount of time between successive updates of the variables that represent time must be exactly one second. Unfortunately, this program was doomed from the start because it does not take into account the time required to update the variables and print the time. The total time required for one iteration of the loop is the sum of the time required to evaluate the Boolean expression, update the variables, and print the current time, plus the 1-second delay for the `sleep()` call. However, even if we carefully adjusted the sleep interval so that the loop lasted 1 second, the program would still lose time.

The program cannot keep accurate time because the `sleep()` method only guarantees that the thread is in the wait queue for the specified period of time; it makes no promises about when the thread is scheduled. When a thread invokes `sleep()`, the thread is placed in the wait queue for the specified interval. When the sleep interval is over, the thread is moved to the ready queue and now must compete with all the other ready threads in the queue. So, even though the thread will be in the wait queue for the specified time interval, more time may pass before the thread is selected for execution. When you consider also that the scheduler may preempt the thread at any time (for example, to run a higher-priority thread), it becomes clear that the program in Figure 11-14 cannot accurately record the passage of time.

In the programs we discuss in this book, it is not important how accurately they maintain the time. Even the clock program in Figure 11-14 is probably good enough to keep track of time for a human being. However, a special type of program, called a **real-time program**, must produce the correct results within an allotted period of time. If the timing constraints imposed by the requirements are not met, the program is considered incorrect. For example, consider the flight control software for the space shuttle, which issues a command for a thruster to burn for a specific period of time. If the thruster does not burn for the correct amount of time, or if there is a delay between the time the command is issued and the burn starts, the shuttle could hurtle out of orbit or burn up in the atmosphere. Standard implementations of Java would not be suitable for this type of real-time programming.

The `Thread` class defines another static method, `yield()`, that causes the currently executing thread to pause temporarily and allow other threads to execute. Like `sleep()`, `yield()` affects only the thread that is currently executing. However, unlike `sleep()`, which puts the process in the wait queue, when the current thread executes `yield()`, the scheduler immediately places the thread back in the ready queue and then selects the next thread to execute. Because the current thread is placed into the ready queue before the next thread is selected, the thread that was just preempted could be selected for execution. The program in Figure 11-15 illustrates the use of the `yield()` method.

```java
/**
 * Use a user thread to perform some complicated, time-consuming
 * computation. The main thread waits for the worker thread
 * using the yield() method.
 */
public class YieldingMain extends Thread {
    private static final int DELAY = 1000; // Milliseconds to sleep
    private static int result = 0;         // Result of computation

    public void run() {
      // Perform a complicated, time-consuming calculation
      // and store the answer in the variable result

      // Code omitted
    }

    /**
     * Create and start thread to perform computation. Wait until
     * the thread has finished before printing the result.
     *
     * @param args command-line arguments (ignored)
     */
    public static void main(String args[]) {
```

continued

```
        // Create and start the thread
        Thread t = new YieldingMain();
        t.start();

        // Wait for thread to terminate
        while ( t.isAlive() ) {
            Thread.yield();
        }

        // Print result
        System.out.println( result );
    }

} // YieldingMain
```

[FIGURE 11-15] Using `yield()`

Although the use of the `sleep()` and `yield()` methods prevented the deadlock that occurred in Figure 11-12, both solutions still require the use of a busy loop. In Figures 11-12 and 11-15, a while loop was used to determine if the user thread was still active, and if it was, either `sleep()` or `yield()` was invoked to give the other thread a chance to execute. Using a while loop to constantly check if the user thread is active is a waste of processor time. It would be much more efficient to pause the execution of the main thread until the user thread terminates. At this point, the main thread could be restarted and safely print the result of the computation.

The `join()` method from the `Thread` class does exactly what we want. A thread blocks after invoking the `join()`method on another thread until that thread terminates. That is, if thread `a` executes the call `b.join()`, `a` is saying that it wants to pause until thread `b` terminates. At that time, `a` should be returned to the ready queue. Note that `join()` is not a static method. The current thread must have a reference to the thread it wants to join. This is different from the other methods discussed in this section. The program in Figure 11-16 illustrates the `join()` method.

```
/**
 * Use a user thread to perform some complicated, time-consuming
 * computation. The main thread waits for the worker thread
 * using the join() method.
 */
public class JoiningMain extends Thread {
    private static int result = 0; // Result of computation
```

continued

```
    public void run() {
      // Perform a complicated, time-consuming calculation
      // and store the answer in the variable result
      // Code omitted
    }

    /**
    ∂* Create and start thread to perform computation. Wait until
    ∂* the thread has finished before printing the result.
    ∂*
    ∂* @param args command-line arguments (ignored)
    ∂*/
    public static void main(String args[]) {
      // Create and start the thread
      Thread t = new JoiningMain();
      t.start();

      // Wait for thread to terminate
      try {
          t.join(); // Pause execution until t dies
      }
      catch ( InterruptedException e ) {}

      // Print result
      System.out.println( result );
    }

} // JoiningMain
```

[FIGURE 11-16] Using `join()`

In Figure 11-16, the while loop was replaced with a statement that invokes `join()` on the user thread. This causes the main thread to be suspended until the user thread terminates. In terms of scheduling, the main thread remains in the wait queue until the user thread has terminated. At that point, it is placed into the ready queue and eventually selected for execution. By using `join()`, we no longer waste processor time checking whether the user thread is active.

The `join()` method has three versions. The no-argument version, illustrated in Figure 11-16, waits indefinitely for the thread to terminate. The two other versions allow you to specify a maximum time that `join()`waits for the termination of the thread. The parameters that specify this wait interval work exactly like those in `sleep()`. If the time period expires, the `join()` method simply returns. You can use the `isAlive()` method to determine if the other thread actually terminated or is still running.

Programs that use multiple threads can have problems when the threads compete for resources. In this section, we saw what can happen in a program when two or more threads compete for the processor. The program in Figure 11-11 illustrated how this competition can result in a program that does not execute as expected. This program was modified using `sleep()`, `yield()`, and `join()` to provide hints to the scheduler so that the program would execute correctly. Synchronization problems can also occur when concurrent threads access common memory locations. In this case, the program typically runs but gives an incorrect result. The next section examines how this situation can arise and explores the language constructs that Java provides to control how threads execute.

11.3 Synchronization

11.3.1 Background

The processor is not the only resource for which concurrent threads compete. Other shared resources such as files, printers, and memory should be used carefully so that concurrent access to them does not have unexpected consequences. In this section, we look at the problems that can occur when two threads attempt to modify a common memory location at the same time.

To begin, look at the code in Figure 11-17, which creates three threads that concurrently attempt to modify the class variable named `common`. Take a moment to try to determine the output this program produces.

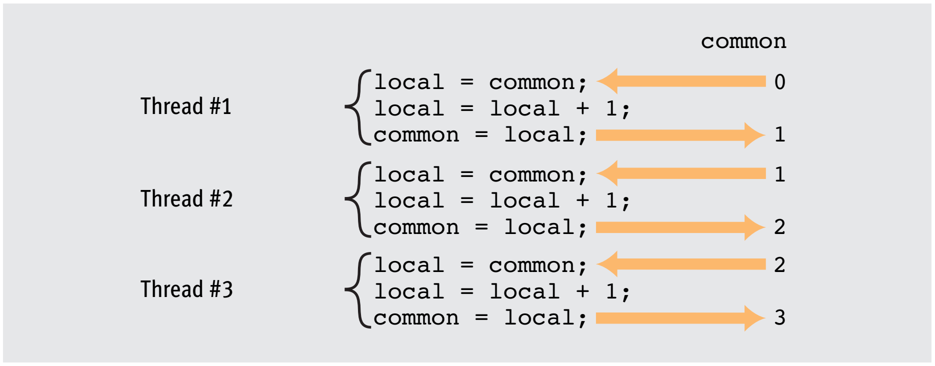
```
/**
 * Add three to a variable in a strange way.
 */
public class ConcAccess extends Thread {
    private static final int NUM_THREADS = 3;
    private static int common = 0;

    /**
     * Obtain a local copy of common, increment the copy by one,
     * and store the result back in common.
     */
    public void run() {
      int local = 0; // Local storage
```

continued

```
        // Add one to common
        local = common;
        local = local + 1;
        common = local;
      }

      /**
       * Create and start three threads that each add one to
       * common. Print common when all threads have died.
       *
       * @param args command-line arguments (ignored).
       */
      public static void main( String args[] ) {
        // Hold the references to the threads that
        // are created so that main can join with them
        Thread myThreads[] = new Thread[ NUM_THREADS ];

        // Create and start the threads
        for ( int i = 0; i < NUM_THREADS; i++ ) {
            myThreads[ i ] = new ConcAccess(); myThreads[ i ].start();
        }

        // Join with each thread
        for ( int i = 0; i < NUM_THREADS; i++ ) {
            try {
              myThreads[ i ].join();
            }
            catch ( InterruptedException e ) {}
        }

        // Threads have terminated; print the result
        System.out.println( "Common is: " + common );
      }

  } // ConcAccess
```

[FIGURE 11-17] Concurrent access

Each thread in this program modifies the class variable `common` by incrementing it by 1. Because the program has three threads, its output should be 3. As it turns out, this program does occasionally produce a wrong answer, such as 1 or 2. The program itself is correct, so we must concentrate on what happens to the threads as they are scheduled and run. Remember that the scheduler is responsible for managing the execution of the three threads and is free to interleave their execution any way it sees fit. Consider what happens if the scheduler executes each thread one at a time until it terminates. The resulting instruction stream is shown in Figure 11-18.

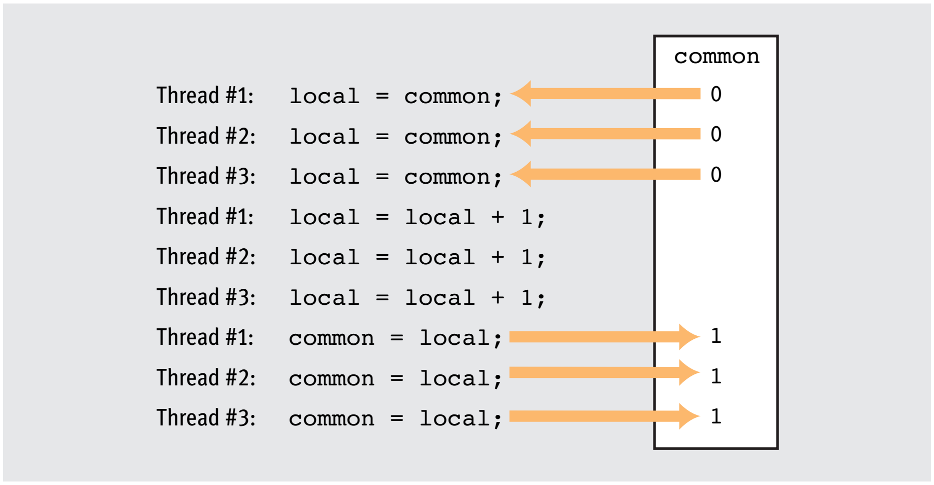


```
local = common;
local = local + 1;
common = local;
local = common;
local = local + 1;
common = local;
local = common;
local = local + 1;
common = local;
```

[FIGURE 11-18] Serial execution

When each thread is run one at a time, the program produces the correct result, which is 3. As you can see in Figure 11-18, each thread obtains a copy of the current value in the variable `common`, increments the local copy, and then stores the result back into `common`. This means that the next thread that accesses `common` will see the updates by all the threads that ran earlier. The order of execution in Figure 11-18 ensures that only one thread is updating `common` at a time. This serializes the execution of the threads and effectively removes any concurrency from the program. When the threads are executed in this way, it is impossible for two threads to update `common` at the same time, and the program produces the correct result.

Now consider what happens if the scheduler runs the threads one statement at a time. In other words, the scheduler executes the first statement from thread 1, the first statement from thread 2, the first statement from thread 3, the second statement from thread 1, and so on. The resulting instruction stream is shown in Figure 11-19.


common
Thread #1: local = common; 0
Thread #2: local = common; 0
Thread #3: local = common; 0
Thread #1: local = local + 1;
Thread #2: local = local + 1;
Thread #3: local = local + 1;
Thread #1: common = local; 1
Thread #2: common = local; 1
Thread #3: common = local; 1

[FIGURE 11-19] Concurrent execution

When the threads are executed in this order, all three threads are updating `common` at the same time. From the start, it is clear that this program will fail because each thread begins by obtaining a copy of the value in `common`. Only one thread sees the "correct" value of `common`; the other threads obtain the value stored in `common` before any of the threads have had a chance to store their results. If the threads are executed one instruction at a time, each thread bases its update on the original value stored in `common`. After the first statements in each thread are executed, it does not matter how the remaining instructions in the threads are executed; the final value stored in `common` is incorrect, and the program prints the wrong answer. In the case of Figure 11-19, it incorrectly stores the value 1.

For this program to work correctly, the threads must be executed so that as soon as a thread retrieves the current value stored in `common`, the remaining threads are forced to wait until the newly updated value has been stored back into this shared variable. The block of code that starts with the statement `local = common` and ends with `common = local` must be executed as a single indivisible unit. As long as any one thread is in the midst of executing this block of code, all other threads must wait for it to finish this block before they can begin to execute. The block of code that updates common is referred to as a **critical section** because for the program to function correctly, it is critical that the code in this section of the program be executed by only one thread at a time.

The program in Figure 11-17 contained only a single critical section; however, a typical multithreaded program may contain many critical sections. For example, consider the code in Figure 11-20. This is the same program shown in Figure 11-17, but an additional class variable, `other`, is incremented by each thread. This program contains two critical sections: one that increments the variable `common` and another that increments the variable `other`.

```
/**
 * A program with two critical sections.
 */
public class TwoCritical extends Thread {
    private static final int NUM_THREADS = 3;
    private static int common = 0, other = 0;

    /**
     * Increment both common and other by one
     */
    public void run() {
      int local = 0; // Local storage

      // Add one to common
      local = common;
      local = local + 1;
      common = local;

      // Add one to other
      local = other;
      local = local + 1;
      other = local;
    }

    /**
     * Create and start three threads.
     *
     * @param args command-line arguments (ignored)
     */
    public static void main( String args[] ) {
      // References to the threads
      Thread myThreads[] = new Thread[ NUM_THREADS ];

      // Create and start the threads
      for ( int i = 0; i < NUM_THREADS; i++ ) {
          myThreads[ i ] = new ConcAccess();
          myThreads[ i ].start();
      }

      // Join with each thread
      for ( int i = 0; i < NUM_THREADS; i++ ) {
          try {
            myThreads[ i ].join();
          }
          catch ( InterruptedException e ) {}
      }
```

continued

```
        // Threads have terminated; print the result
        System.out.println( "Common: " + common + "Other: " + other );
      }

  } // TwoCritical
```

[FIGURE 11-20] Two critical sections

For the program in Figure 11-20 to function correctly, you must make sure that only one thread is executing within each critical section. However, it would be acceptable for one thread to update `other` while another updated `common`. A program fails if more than one thread attempts to update the same shared variable at the same time. Two threads cannot be in the same critical section at the same time, but two threads can be in two different critical sections at the same time without causing a problem.

When writing multithreaded programs, you must identify all of its critical sections and take action to ensure that multiple threads do not attempt to execute the same critical section of code at the same time. It is often difficult to spot all of the critical sections in a program. Errors that result from concurrent access to shared variables are among the most common mistakes in deployed software. You can typically identify critical sections by looking for code segments that modify shared resources such as variables, objects, or files. Although we have concentrated on concurrent access to variables in this section, all of the issues just discussed apply to almost any shared resource in a program.

Once the critical sections are identified, we can take appropriate steps to ensure that only one thread executes this code at a time. Java provides a locking mechanism that allows a programmer to inform the scheduler that a particular section of code can be executed by only one thread at a time. By ensuring exclusive access to this critical section, we can be much more confident that programs work as intended.

THE THERAC-25 ACCIDENTS

Starting in 1976, the Therac-25 treatment system was used to fight cancer by providing radiation to a specific part of the body in the hope of destroying tumors. In 1982, a cancer patient received burns from a Therac-25 that administered a large overdose of radiation. As a result, she had to have both breasts removed and lost use of her right arm. In 1985, a second patient received about 600 times more than the normal dose of radiation and died of cancer three months later. Six known Therac-25 accidents have been documented;

continued

all involved massive overdoses of radiation and resulted in patient death or serious injury. Patients received an estimated 17,000 to 25,000 rads to very small body areas. By comparison, doses of 1000 rads can be fatal if delivered to the whole body.

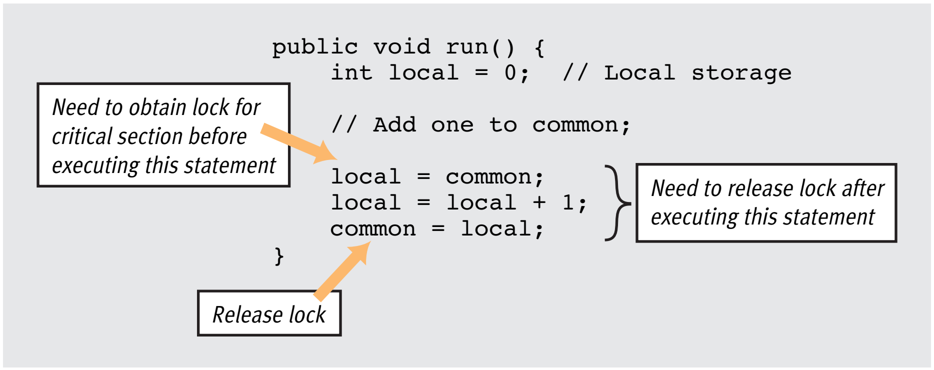
Analysis determined that the primary cause of the overdoses was faulty software. The software was written in assembly language and was developed and tested by the same person. The software included a scheduler and concurrency in its design. When the system was first built, operators complained that it took too long to enter the treatment plan into the computer. As a result, the software was modified to allow operators to quickly enter treatment data by simply pressing the Enter key when an input value did not require changing. This change created a synchronization error between the code that read the data entered by the operator and the code controlling the machine. As a result, the actions of the machine would lag behind commands the operator entered. The machine appeared to administer the dose entered by the operator, but in fact had an improper setting that focused radiation at full power to a tiny spot on the body. The basic cause of this error was not very different from the error illustrated in Figure 11-17.

C. A. R. Hoare, who is profiled at the end of this chapter, made the following comment in his 1980 ACM Turing Award lecture, entitled “The Emperor’s Old Clothes”: “An unreliable programming language generating unreliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations. Be vigilant to reduce that risk, not to increase it.”

11.3.2 Locks

Every object in a Java program, whether it is defined by a user or a class in a library, is associated with a **lock** that provides a means of synchronization. Only one thread can hold a given lock at one time. If two threads try to obtain the same lock at the same time, only one obtains it. The other thread must wait until the first thread releases the lock before it can obtain the lock. This type of lock is called a **mutually exclusive**, or **mutex**, lock because access to it is exclusive.

Given the previous discussion about critical sections, it is not difficult to see how a mutex lock could ensure that only one thread can be in a critical section at a time. We first associate a lock with a specific critical section. When a thread wants to enter a critical section, it must obtain the lock for that block of code before it can execute the code. After the thread finishes executing the code in the critical section, it releases the lock to inform other threads that might be waiting to enter the section that they may now attempt to obtain the lock and enter the critical section. This process is illustrated in Figure 11-21.

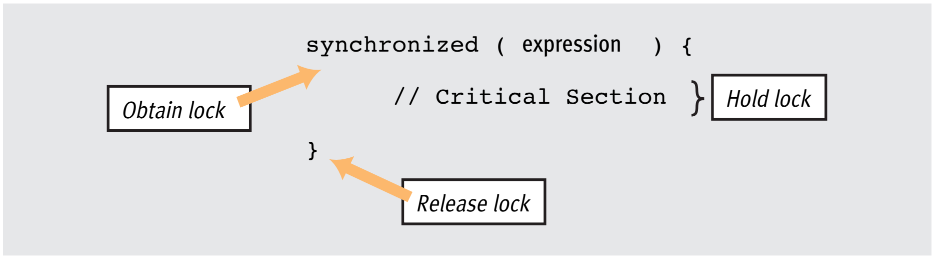


```
public void run() {
    int local = 0;   // Local storage

    // Add one to common;

    local = common;
    local = local + 1;
    common = local;
}
```

Need to obtain lock for critical section before executing this statement
Need to release lock after executing this statement
Release lock

[FIGURE 11-21] Using a mutex lock

The Java programming language provides a syntactic construct called a **synchronized block** to control the access to critical sections of a program. The syntax of the `synchronized` statement, which is used to identify synchronized blocks, is shown in Figure 11-22.



```
synchronized ( expression  ) {

      // Critical Section

}
```

[FIGURE 11-22] A `synchronized` statement

Before executing any code in a synchronized block, the thread must obtain the lock associated with the object specified by `expression` in Figure 11-22. The thread holds the lock as long as it is executing the code inside the synchronized block.

When the thread executes the last statement in the block and leaves the block, the lock is released. Because only one thread can hold a specific lock at a time, only one thread can be in the critical section. If a thread cannot obtain the lock when it attempts to enter the synchronized block, it must wait until the lock is available. More than one thread can wait for a lock. When the lock is released, the JVM selects a waiting thread, gives it the lock, and allows it to enter the block. The order in which the waiting threads are selected to obtain the lock is determined by the JVM at run time. Figure 11-23 contains a modified fragment of the program from Figure 11-17 that illustrates the use of the `synchronized` statement.

```
public class ConcAccess extends Thread {
    private static final int NUM_THREADS = 3;
    private static int common = 0;

    private static final Integer lock = new Integer( 0 );

    public void run() {
      int local = 0; // Local storage

      // Add one to common
      synchronized( lock ) {
          local = common;
          local = local + 1;
          common = local;
      }

    }

    // Rest of program omitted
```

[FIGURE 11-23] Using `synchronized` in a program

In Figure 11-23, a static `Integer` object controls access to the synchronized block, although the lock can be any type of object. The key to making the synchronized block work correctly is to make sure that all threads attempt to lock the same object when entering the block. If the threads obtain locks on different objects, multiple threads are allowed to enter the synchronized block. This is why the variable lock is declared static in Figure 11-23. Regardless of how many threads `main()` creates, they all attempt to obtain the same static variable lock when entering the synchronized block. The code in Figure 11-24 also arranges to have all of the threads lock on the same object. In this case, however, the reference to the lock is stored in an instance variable.

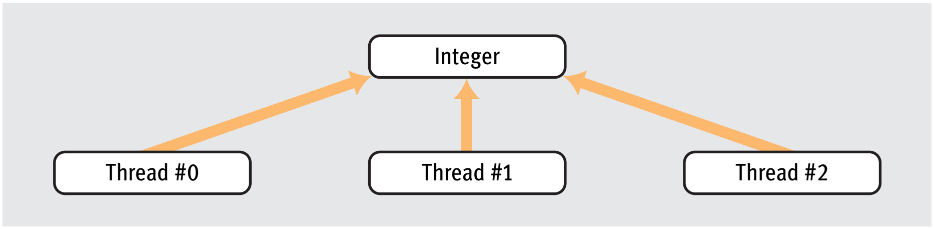
```
/**
 * Demonstrate object locking in Java.
 */
public class Locks1 extends Thread {
    private static final int NUM_THREADS = 3;         // Num of threads
    private static final String PREFIX = "Thread #"; // For thread name
    private static final int N = 3;                   // Loop constant
    private static final int DELAY = 10;              // Sleep period

    private Object lock; // Object used for synchronization
```

continued

```
    public Locks1( Object l, String name ) {
        // Store a reference to the lock and the name of the thread
        lock = l;
        setName( name );
    }

    public void run() {
        // Get the lock before entering the loop
        synchronized( lock ) {
            for ( int i = 0; i < N; i++ ) {
                System.out.println( getName() + " is tired" );

                try {
                    Thread.currentThread().sleep( DELAY );
                }
                catch ( InterruptedException e ){}

                System.out.println( getName() + " is rested" );
            }
        }
    }

    public static void main( String args[] ) {
        // Create the object to use as the lock
        Integer lock = new Integer( 0 );

        // Create the threads and let them run
        for ( int i = 0; i < NUM_THREADS; i++ ) {
            new Locks1( lock, PREFIX + i ).start();
        }
    }
} // Locks1
```

[FIGURE 11-24] Synchronizing on a common lock

The critical section in Figure 11-24 is the for loop in the `run()` method that executes three times. Each time the loop runs, it prints a message that the thread is tired, sleeps for 10 milliseconds, and then prints a message that the thread is rested. Because this loop is inside a synchronized block, you might be tempted to say that once any thread starts executing the loop, all the other threads must wait for the lock to be released. This occurs only if all of the threads attempt to obtain the lock on the same object. Looking at the `main()` method and the constructor for the class, you can see that only one object is being used as a lock in this program. The `main()` method creates an `Integer` object and then passes a reference to that object to each of the threads it creates. The diagram in Figure 11-25 shows the object that each thread uses as a lock. Because the instance variables all refer to the same object, only one thread can execute the for loop at a time.



[FIGURE 11-25] Threads and their locks

Although we have determined that the for loop is executed by each thread exclusively, we still cannot be completely certain about the behavior (output) of this program because we do not know the order in which the threads are executed. For example, assume for a moment that all three threads reach the synchronized statement at exactly the same time. Only one thread (we do not know which one) can obtain the lock, while the other two must wait. When the first thread leaves the loop, one of the two waiting threads can obtain the lock, but again we do not know which one. So, even though the `synchronized` statement made it possible to ensure that only one thread was executing the for loop at any one time, we still cannot determine the exact output of this program. Figure 11-26 shows three of the six possible outputs you can generate with this program.

```
Thread #0 is tired
Thread #0 is rested
Thread #0 is tired
Thread #0 is rested
Thread #0 is tired
Thread #0 is rested
Thread #1 is tired
Thread #1 is rested
Thread #1 is tired
Thread #1 is rested
Thread #1 is tired
Thread #1 is rested
Thread #2 is tired
Thread #2 is rested
Thread #2 is tired
Thread #2 is rested
Thread #2 is tired
Thread #2 is rested
```

```
Thread #1 is tired
Thread #1 is rested
Thread #1 is tired
Thread #1 is rested
Thread #1 is tired
Thread #1 is rested
Thread #0 is tired
Thread #0 is rested
Thread #0 is tired
Thread #0 is rested
Thread #0 is tired
Thread #0 is rested
Thread #2 is tired
Thread #2 is rested
Thread #2 is tired
Thread #2 is rested
Thread #2 is tired
Thread #2 is rested
```

```
Thread #2 is tired
Thread #2 is rested
Thread #2 is tired
Thread #2 is rested
Thread #2 is tired
Thread #2 is rested
Thread #1 is tired
Thread #1 is rested
Thread #1 is tired
Thread #1 is rested
Thread #1 is tired
Thread #1 is rested
Thread #0 is tired
Thread #0 is rested
Thread #0 is tired
Thread #0 is rested
Thread #0 is tired
Thread #0 is rested
```

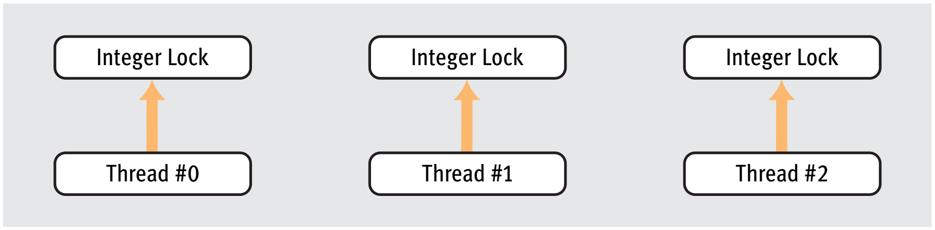
[FIGURE 11-26] Three of six possible outputs

Now consider the program in Figure 11-27. It is almost identical to the program in Figure 11-24, except for how the lock is created and distributed to the threads. The integer lock is created in the constructor instead of being created in `main()` and distributed to the threads as a parameter.

```
/**
 * Demonstrate object locking in Java.
 */
public class Locks2 extends Thread {
    private static final int NUM_THREADS = 3;        // Num of threads
    private static final String PREFIX = "Thread #"; // For thread name
    private static final int N = 3;                  // Loop constant
    private static final int DELAY = 10;             // Sleep period

    private Object lock;  // Object used for synchronization

    public Locks2( String name ) {
        // Store a reference to the lock and the name of the thread
        lock = new Integer( 0 );
        setName( name );
    }

    public void run() {
        // Get the lock before entering the loop
        synchronized( lock ) {
            for ( int i = 0; i < N; i++ ) {
                System.out.println( getName() + " is tired" );

                try {
                    Thread.currentThread().sleep( DELAY );
                }
                catch ( InterruptedException e ){}

                System.out.println( getName() + " is rested" );
            }
        }
    }

    public static void main( String args[] ) {
        // Create the threads and let them run
        for ( int i = 0; i < NUM_THREADS; i++ ) {
            new Locks2( PREFIX + i ).start();
        }
    }

} // Locks2
```

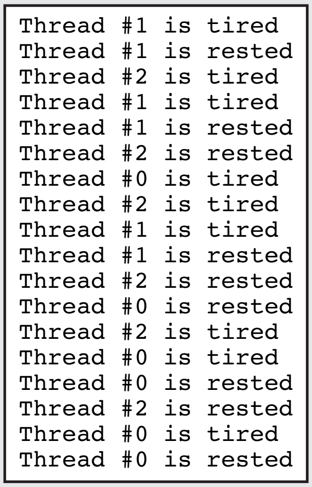
[FIGURE 11-27] Attempted synchronization using an instance variable

This small change in Figure 11-27 dramatically changes how the program runs. The for loop is no longer executed by the threads one at a time. Instead, all three threads can execute the loop at the same time. This is puzzling because the `run()` method has not changed from the previous program. The for loop is still in a synchronized block, and a lock still must be obtained to execute the loop. The difference between this program and the program in Figure 11-24 lies in the locks. In Figure 11-27, each thread creates its own lock when the constructor is called. So, as shown in Figure 11-28, each thread has a reference to a different lock. The `synchronized` statement still obtains the lock before executing the loop, but because each thread has a different lock, they can all execute the for loop at the same time. The synchronization in the program of Figure 11-24 has been lost.



[FIGURE 11-28] Different lock objects

Because every object is locking on different locks, there is no synchronization. The `synchronized` statement still obtains and releases the lock as we have described, but each thread uses a different lock to control access to the loop. The output of this program consists of the six lines of text printed by each thread interleaved in random order, as shown in Figure 11-29.



```
Thread #1 is tired
Thread #1 is rested
Thread #2 is tired
Thread #1 is tired
Thread #1 is rested
Thread #2 is rested
Thread #0 is tired
Thread #2 is tired
Thread #1 is tired
Thread #1 is rested
Thread #2 is rested
Thread #0 is rested
Thread #2 is tired
Thread #0 is tired
Thread #0 is rested
Thread #2 is rested
Thread #0 is tired
Thread #0 is rested
```

[FIGURE 11-29] Random output resulting from multiple locks

One way to ensure that only one object is being used as a lock in Figure 11-27 is to use a class variable as a lock. Because a class variable has only one copy, all of the instances of the class use the same object as a lock. The program in Figure 11-30 uses the class variable `lock` to synchronize the actions of the threads it creates.

```java
/**
 * Use a static lock for synchronization
 */
public class Locks3 extends Thread {
    private static final int NUM_THREADS = 3;         // Num of threads
    private static final String PREFIX = "Thread #"; // For thread name
    private static final int N = 3;                   // Loop constant
    private static final int DELAY = 10;              // Sleep period

    // Object used for synchronization
    private static final Object lock = new Integer( 0 );

    public Locks3( String name ) {
        // Store the name of the thread
        setName( name );
    }

    public void run() {
        // Get the lock before entering the loop
        synchronized( lock ) {
            for ( int i = 0; i < N; i++ ) {
                System.out.println( getName() + " is tired" );

                try {
                    Thread.currentThread().sleep( DELAY );
                }
                catch ( InterruptedException e ){}

                System.out.println( getName() + " is rested" );
            }
        }
    }

    public static void main( String args[] ) {
        // Create the threads and let them run
        for ( int i = 0; i < NUM_THREADS; i++ ) {
            new Locks3( PREFIX + i ).start();
        }
    }

} // Locks3
```

[FIGURE 11-30] Using a static lock

As you learned earlier, every object in Java has a lock and any expression that evaluates to a reference to an object can be used in a `synchronized` statement. Consider the code in Figure 11-31, which uses the `this` variable associated with an object to obtain a reference to the object whose lock controls access to the synchronized block.

```
/**
 * Using this to lock in Java.
 */
public class ThisLock extends Thread {
    private static final int NUM_THREADS = 3;        // Num of threads
    private static final String PREFIX = "Thread #"; // For thread name
    private static final int N = 3;                  // Loop constant
    private static final int DELAY = 10;             // Sleep period

    public ThisLock( String name ) {
      setName( name );
    }

    public void run() {
      // Get the lock before entering the loop
      synchronized( this ) {
          for ( int i = 0; i < N; i++ ) {
            System.out.println( getName() + " is tired" );

            try {
                Thread.currentThread().sleep( DELAY );
            }
            catch ( InterruptedException e ){}

            System.out.println( getName() + " is rested" );
          }
      }
    }

    public static void main( String args[] ) {
      // Create the threads and let them run
      for ( int i = 0; i < NUM_THREADS; i++ ) {
          new ThisLock( PREFIX + i ).start();
      }
    }

} // ThisLock
```

[FIGURE 11-31] Attempted synchronization using `this`

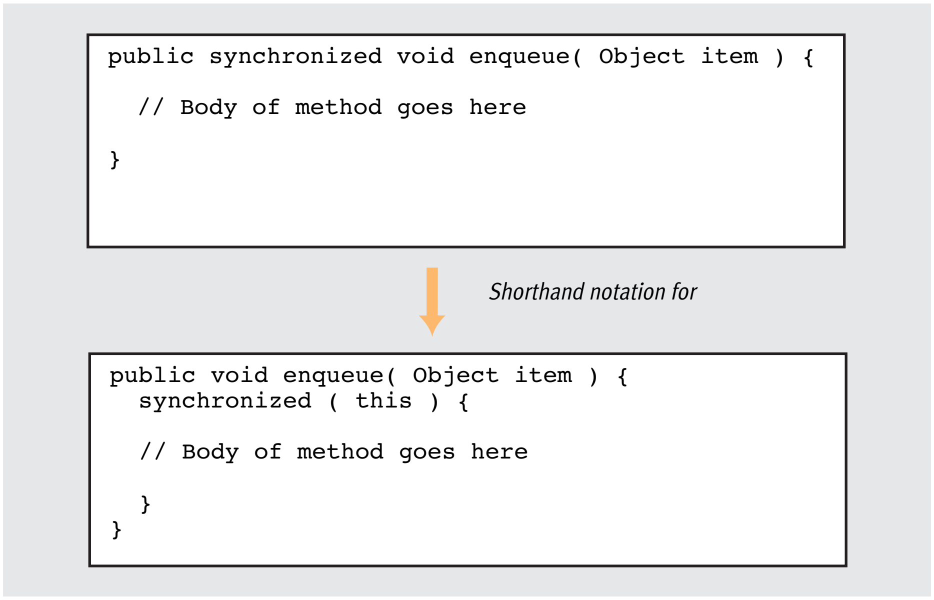
Using `this` in Figure 11-31 causes the program to behave exactly like the program in Figure 11-27. The program in Figure 11-31 has no synchronization because each thread is locking on a different object. In this case, instead of locking on different `Integer` objects, the threads lock on different `ThisLock` objects. At first glance, writing a method that locks on the object in which it is defined seems useless. However, consider the code fragment for the `Queue` class shown in Figure 11-32.

```
/**
 * A thread-safe queue class.
 */
public class SyncQueueClass {
    // Instance variables go here

    public void enqueue( Object item ) {
        synchronized ( SyncQueueClass ) {
            // Code to enqueue goes here
        }
    }

    public Object dequeue() {
        synchronized ( SyncQueueClass ) {
            // Code to dequeue goes here
        }
    }

} // SyncQueueClass
```

[FIGURE 11-32] Thread-safe queue

In Chapter 7, you learned that the `enqueue()`method adds items to a queue and the `dequeue()`method removes items from a queue. Clearly, these two methods should never be executed at the same time by two different threads because the methods access the shared state of the `Queue` object (in other words, `size`, `front`, and `back`). Unpredictable results could occur if an `enqueue()` and `dequeue()` operation occurred at the same time. By writing the `enqueue()` and `dequeue()` methods so that they have to obtain the lock on the `Queue` object before execution, you solve this problem. If you try to execute an `enqueue()` and `dequeue()`method at the same time, only one thread can obtain the object lock. The other thread must wait until the first thread has released the lock.

Synchronizing methods in this way is so useful that Java allows you to define a method as synchronized. **Synchronized methods** are nothing more than a shorthand notation for a synchronization block that surrounds the code in the method, as shown in Figure 11-33. The JVM must obtain the lock associated with the object on which the method is invoked before the statements in the instruction can be executed.



```
public synchronized void enqueue( Object item ) {

  // Body of method goes here

}
```

```
public void enqueue( Object item ) {
  synchronized ( this ) {

  // Body of method goes here

  }
}
```

[FIGURE 11-33] Synchronized methods

When using synchronized methods, keep in mind that the lock that controls access to these methods is the object itself. This means there is synchronization within the object; only one synchronized method in the object can execute at one time. The shorthand notation does not mean that the synchronization holds across all instances of the class. For example, in the code in Figure 11-33, only one thread at a time can place objects into or remove objects from any one queue. However, objects can be placed in different queue objects at the same time.

Whenever the JVM loads a class, it automatically instantiates a `Class` object that represents the class. You can use a `Class` object to obtain class information, such as its full name, the constructors it provides, or its associated methods. (For the complete specification of the class `Class`, refer to the appropriate Javadoc page in the Java API.) An instance of a class can obtain a reference to its `Class` object by invoking the `getClass()` method defined in class `Object`, as illustrated in Figure 11-34.

```
/**
 * Use the class Class to obtain basic information about the
 * class Integer.
 */
```

continued

```
public class UsingClass {
    public static void main( String args[] ) {
        // Create an instance of an integer
        Integer anInt = new Integer( 0 );

        // Obtain a reference to the class object associated
        // with an integer
        Class aClass = anInt.getClass();

        // Print the name of the class and the package using
        // accessors provided by the Class object
        System.out.println( "Class:   " + aClass.getName() );
        System.out.println( "Package:   " + aClass.getPackage() );
    }

}  // UsingClass
```

[FIGURE 11-34] Using the class `Class`

For every class loaded by the JVM, only one `Class` object is created to represent the class, regardless of the number of instances of the class a program might have. Thus, when any instance of a particular class invokes the `getClass()` method, a reference to the same object is always returned. Because an object is used to represent the class and every object has an associated lock, there is no reason instances of a class could not use the instance of the `Class` object returned by `getClass()` for synchronization. Figure 11-35 illustrates how you could modify the program in Figure 11-30 to use its `Class` object as a lock instead of a class variable.

```
/**
 * Use a class object for synchronization
 */
public class Locks4 extends Thread {
    private static final int NUM_THREADS = 3;        // Num of threads
    private static final String PREFIX = "Thread #"; // For thread name
    private static final int N = 3;                  // Loop constant
    private static final int DELAY = 10;             // Sleep period
    public Locks4( String name ) {
      // Store the name of the thread
      setName( name );
    }
```

continued

```
    public void run() {
      // Get the lock before entering the loop
      synchronized( getClass() ) {
          for ( int i = 0; i < N; i++ ) {
            System.out.println( getName() + " is tired" );

            try {
                Thread.currentThread().sleep( DELAY );
            } catch ( InterruptedException e ){}

            System.out.println( getName() + " is rested" ); }
      }
    }

    public static void main( String args[] ) {
      // Create the threads and let them run
      for ( int i = 0; i < NUM_THREADS; i++ ) {
          new Locks4( PREFIX + i ).start();
      }
    }

} // Locks4
```

[FIGURE 11-35] Using a `Class` object for synchronization

Figure 11-33 illustrates that when you invoke a synchronized method, the JVM uses it to obtain a lock before any of the method code is executed. You can declare static methods as synchronized, as shown in Figure 11-36. In a static synchronized method, the JVM obtains the lock associated with the object returned by `getClass()` before executing the method code. A static method cannot use `this` to obtain a reference to a lock because static methods do not have a `this` reference.

```
/**
 * Static synchronized methods
 */
public class Locks5 extends Thread {
    private static final int NUM_THREADS = 3;         // Num of threads
    private static final String PREFIX = "Thread #"; // For thread name
    private static final int N = 3;                   // Loop constant
    private static final int DELAY = 10;              // Sleep period

    public Locks5( String name ) {
      // Store the name of the thread
      setName( name );
    }
```

continued

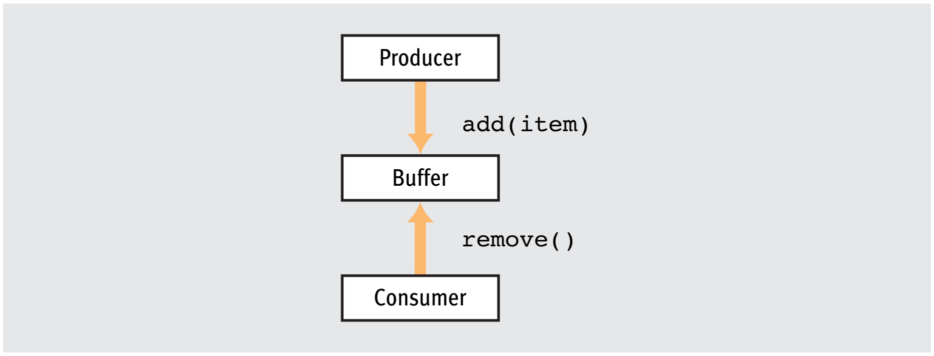
```
    public static synchronized void doIt( String name ) {
      for ( int i = 0; i < N; i++ ) {
          System.out.println( name + " is tired" );

          try {
            Thread.currentThread().sleep( DELAY );
          }
          catch ( InterruptedException e ){}

          System.out.println( name + " is rested" );
      }
    }

    public void run() {
        doIt( getName() );
    }

    public static void main( String args[] ) {
        // Create the threads and let them run
        for ( int i = 0; i < NUM_THREADS; i++ ) {
            new Locks5( PREFIX + i ).start();
        }
    }

} // Locks5
```

[FIGURE 11-36] Static synchronized methods

This section discussed how to use locks to synchronize the behavior of concurrent threads in a Java program. Every object in a Java program has an associated lock; by using a synchronized block, you can obtain a lock before executing the code within the block. Because a lock can be held by only one thread at a time, only one thread can execute the code in a synchronized block. Locks and synchronized blocks provide the basis for synchronization in a Java program; however, as the next section shows, additional tools are needed to solve the synchronization issues faced by multithreaded programs.

11.3.3 The `wait()` and `notify()` Methods

Although the synchronization primitives we have discussed so far allow you to coordinate the activities of different threads, additional capabilities are sometimes required. To illustrate this, consider the interactions between a producer of items and a consumer of the items. The producer and consumer run as separate threads and are connected by a buffer. The producer places the items in the buffer and the consumer removes them. The structure of this problem is shown in Figure 11-37.



[FIGURE 11-37] Producer/consumer problem

The `Buffer` class provides two methods: `add()` and `remove()`. The producer invokes `add()` to place an item in the buffer, and the consumer invokes `remove()` to retrieve the item. Clearly, the code that implements `add()` and `remove()` must be written so that one thread cannot add an item while a second thread is removing an item at the same time. In terms of the producer/consumer problem, we do not want to allow a producer to execute the `add()` method at the same time the consumer executes `remove()`. Additionally, `add()` must work correctly if it is invoked when the buffer is full, and `remove()` must work correctly if it is invoked on an empty buffer.

The interface in Figure 11-38 defines the behavior associated with a finite `Buffer`. The interface requires that a buffer provide two methods: `add()` and `remove()`. The method `add()` blocks until the item has been added to the buffer. In other words, if `add()` is invoked on a full buffer, the method does not return until the item has been added to the buffer. If the buffer is full when `add()` is invoked, the method waits until the buffer has room for the additional item. Likewise, if `remove()` is invoked on an empty buffer, the method waits until an item is available for removal. So, if a consumer invokes `remove()` and no item is available, it waits until the producer has placed an item in the buffer by invoking `add()`.

```
/**
 * A buffer that can serve as a connection between a
 * producer and a consumer.
 */
public interface Buffer {
    /**
     * Add an element to the buffer. If the buffer is full,
     * the method blocks until the item can be placed in
     * the buffer.
     *
```

continued

```
     * @param item the element to add to the buffer
     */
    public void add( int item );

    /**
     * Remove an element from the buffer. If the buffer is empty,
     * the method blocks until an item is available.
     *
     * @returns the next item in the buffer
     */
    public int remove();

} // Buffer
```

[FIGURE 11-38] Buffer interface

Given the constraint that at most one thread can execute `add()` or `remove()` in a single instance of a buffer at one time, it seems reasonable to implement these methods as synchronized methods (see Figure 11-39).

```
/**
 * An implementation of a buffer that can lead to deadlock.
 */
public class DeadlockBuffer implements Buffer {
    private int currentItem;  // Item currently in the buffer
    private boolean full;     // Is the buffer full?

    /**
     * Create a new buffer.
     */
    public DeadlockBuffer() {
      full = false;
    }

    /**
     * Add an element to the buffer. If the buffer is full,
     * the method blocks until the item can be placed in
     * the buffer.
     *
     * @param item the element to add to the buffer
     */
    public synchronized void add( int item ) {
      // Wait until there is room in the buffer for the item
      while ( full );
```

continued

```
        currentItem = item;
        full = true;
      }

      /**
       * Remove an element from the buffer. If the buffer is empty,
       * the method blocks until an item is available.
       *
       * @returns the next item in the buffer
       */
      public int remove() {
        // Wait until there is something to remove
        while ( !full );

        full = false;
        return currentItem;
      }

  } // DeadlockBuffer
```

[FIGURE 11-39] Implementation of a buffer

Although the code in Figure 11-39 satisfies the synchronization constraints of this program (in other words, `add()` and `remove()` cannot be executed at the same time), it has a fatal flaw: deadlock is possible. Consider what would happen if the consumer invoked the `remove()` method on an empty buffer. The consumer would obtain the lock associated with the buffer and execute the code in the `remove()` method. Because the buffer is empty, the consumer would execute the while loop until there is an item to remove (in other words, the buffer is no longer empty). However, as long as the consumer is executing the `remove()` method and holding the lock, the producer cannot obtain the lock and place an item in the buffer. The consumer does not give up the lock until the producer adds something to the buffer. The result is a deadlock. A similar situation can arise if a producer fills the buffer before a consumer has a chance to remove an item.

You can implement a buffer using only locks. The key is to write the code so that when either the consumer or the producer is waiting, it does not maintain exclusive control of the lock. The code in Figure 11-40 contains an implementation of a buffer that uses this technique.

```
  /**
   * An implementation of the producer/consumer buffer
   * that uses simple synchronization.
   */
```

continued

```
public class SyncBuffer implements Buffer {
    private int currentItem;  // Item currently in the buffer
    private boolean full;     // Is the buffer full?

    /**
     * Create a new buffer.
     */
    public SyncBuffer() {
      full = false;
    }

    /**
     * Add an element to the buffer. If the buffer is full,
     * the method blocks until the item can be placed in
     * the buffer.
     *
     * @param item the element to add to the buffer
     */
    public void add( int item ) {
      boolean added = false;

      while ( !added ) {
          synchronized( this ) {
            if ( !full ) {
                currentItem = item;
                full = true;
                added = true;

            }
          }

          Thread.yield();
      }
    }

    /**
     * Remove an element from the buffer. If the buffer is empty,
     * the method blocks until an item is available.
     *
     * @returns the next item in the buffer
     */
    public int remove() {
      int retVal = 0;
      boolean removed = false;
```

continued

```
        while ( !removed ) {
            synchronized( this ) {
              if ( full ) {
                  retVal = currentItem;
                  full = false;
                  removed = true;
              }
            }

            Thread.yield();
        }

        return retVal;
      }

  } // SyncBuffer
```

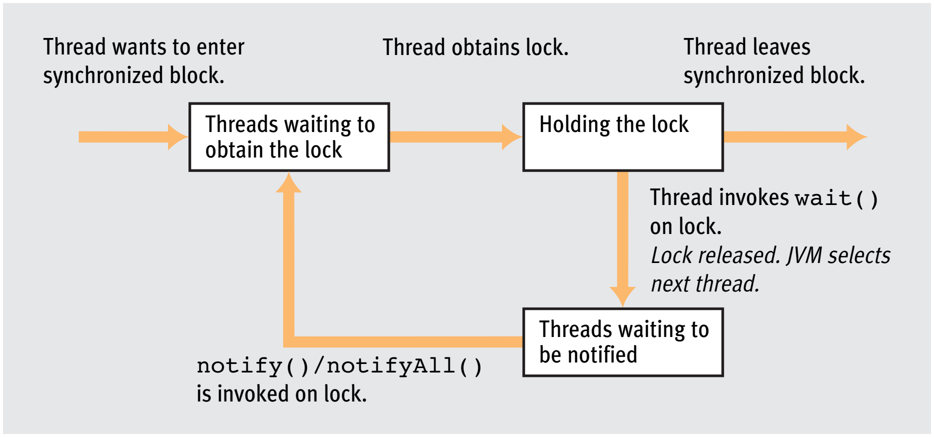
[FIGURE 11-40] A synchronized buffer

The `add()` method in Figure 11-40 has not been written as a synchronized method; instead, it uses a synchronized block to obtain the lock only when it needs to look at the state of the buffer. If the item has not been added to the buffer, the method obtains the lock on the buffer and checks to see if the buffer has room for the item. If the buffer has room, the lock is released and the thread invokes `yield()` to allow the consumer to check the buffer and remove the element. This implementation is not completely deadlock-free because Java does not guarantee that `yield()` will force the JVM scheduler to perform a context switch and run a different thread.

The implementation of the buffer in Figure 11-40 is not very efficient because both the `add()` and `remove()` methods use busy wait loops to check the status of the buffer. The processor spends most of its time executing these loops, waiting for items to be either produced or consumed. It would be much more efficient if the consumer, upon realizing that nothing is in the buffer, suspended itself until the producer places the next item in the buffer.

In Java, a thread that is holding a lock can pause its execution and release the lock. When a thread invokes `wait()` on the lock it is holding, execution of the thread is suspended, the thread is placed in the wait set for the lock, and the lock held by the thread is released. An object has two associated sets that it uses to manage its lock: a set that contains the threads waiting to obtain the object's lock and a set containing the threads that are waiting on the object's lock. A thread that has been suspended is placed in the set with the other threads waiting on the lock and remains there until another thread invokes `notify()` or `notifyAll()` on the lock.

When a thread invokes `notify()`, the JVM removes one of the threads from the wait set of the lock and places the thread in the set that contains the threads waiting to obtain the lock. There is no way to select which thread is removed. Also, after the thread is released, it still has to obtain the lock to resume its execution. The thread that was released is treated no differently than the other threads trying to obtain the lock. The only difference is that once the released thread obtains the lock, it resumes execution at the point it was suspended, as if the `wait()` call had simply returned. The `notifyAll()` is very similar to `notify()`, except that all of the threads waiting on the lock are notified instead of just one. Invoking `notifyAll()` does not cause more than one thread to enter the synchronized block, because each thread must still obtain the lock before resuming execution. The diagram in Figure 11-41 illustrates how the locking mechanism works.


Thread wants to enter synchronized block.
Thread obtains lock.
Thread leaves synchronized block.
Threads waiting to obtain the lock
Holding the lock
Thread invokes wait() on lock.
Lock released. JVM selects next thread.
Threads waiting to be notified
notify()/notifyAll() is invoked on lock.

[FIGURE 11-41] The `wait()` and `notify()` methods

By looking at the Javadoc page for class `Object`, you can see that `wait()` is overloaded. The alternative versions of `wait()` allow you to specify the maximum time that a thread is willing to wait before being notified. If the timeout period expires before the thread is notified, the thread is automatically removed from the set of waiting threads and placed in the queue, where it waits until it obtains the lock again. If you specify a timeout when invoking `join()`, you are notified if the method is returned because the timeout period expired. If you need to know that the timeout period expired, you must provide additional logic in your program to determine when it happens. A thread cannot invoke `wait()`, `notify()`, or `notifyAll()` on a lock unless it is currently holding the lock.

You can use the `wait()` and `notify()`methods to provide the synchronization required in the buffer that connects the producer and the consumer. Both the `remove()` and `add()` methods in the buffer would be written as synchronized methods. This time, instead of using a loop to constantly check the status of the buffer, `wait()` is used to suspend the execution of the thread. In the `remove()` method, if the buffer is empty, the consumer invokes `wait()` on the lock it is holding. This suspends the execution of the consumer so that it is no longer using the processor and releases the lock so that the producer can add an element to the buffer. After adding the item to the buffer, the producer invokes `notify()` to tell the consumer it can safely remove the item and consume it. An implementation of the buffer using `wait()` and `notify()` is shown in Figure 11-42.

```
/**
 * An implementation of the producer/consumer buffer
 * that uses wait and notify.
 */
public class MonitorBuffer implements Buffer {
    private int currentItem;  // Item currently in the buffer
    private boolean full;     // Is the buffer full?

    /**
     * Create a new buffer.
     */
    public MonitorBuffer() {
      full = false;
    }

    /**
     * Add an element to the buffer. If the buffer is full,
     * the method blocks until the item can be placed in
     * the buffer.
     *
     * @param item the element to add to the buffer
     */
    public synchronized void add( int item ) {
      // If the buffer is full, wait until the consumer has removed
      // the element currently in the buffer.

      if ( full ) {
          try {
            wait();
          }
          catch ( InterruptedException e ) {};
      }

      // Place the item in the buffer
      currentItem = item;
      full = true;

      // A consumer might be waiting for an element to consume
      notify();
    }
```

continued

```
    /**
     * Remove an element from the buffer. If the buffer is empty,
     * the method blocks until an item is available.
     *
     * @returns the next item in the buffer
     */
    public synchronized int remove() {
      int retVal;

      // If the buffer is empty, wait until notified by the producer
      // that there is something to consume.
      if ( !full ) {
          try {
            wait();
          }
          catch ( InterruptedException e ) {};
      }

      // Remove the current element
      retVal = currentItem;
      full = false;

      // A producer might be waiting to produce.
      notify();

      // Return the item
      return retVal;
    }

} // MonitorBuffer
```

[FIGURE 11-42] Using `wait()` and `notify()` to implement a buffer

Using `wait()` and `notify()`, you can specify the order in which threads should be granted access to a synchronized block of code. For example, consider writing a program that models the behavior of customers and a cashier at a store. Our implementation consists of two classes: `Customer` and `Cashier`. Each customer in the store must use an instance of a `Cashier` to pay for the items they want. A cashier serves customers one at a time. The interface in Figure 11-43 defines the behavior a `Cashier` must provide.

```
/**
 * A cashier at a supermarket. When customers are ready to check out,
 * they invoke readyToCheckOut(). The cashier then determines
 * which customer to serve. When the current customer
```

continued

```
 * finishes checking out, it invokes done().
 */
public interface Cashier {
    /**
     * Invoked by a customer who is ready to check out.
     * Contains the logic required to select the one customer
     * the cashier will serve. If the cashier is
     * serving another customer, the next customer waits until
     * the first one has finished with the cashier.
     */
    public void readyToCheckOut();

    /**
     * Invoked by a customer who is finished with the
     * cashier. If customers are waiting for the cashier,
     * one is selected and served by the cashier.
     */
    public void done();

} // Cashier
```

[FIGURE 11-43] Interface for a `Cashier`

Given the interface for a `Cashier`, you can write the code that implements a customer. Each customer runs in a separate thread and randomly selects the number of items to purchase. The customer invokes the `readyToCheckOut()` method on a cashier to indicate that he is ready to check out. When the method returns, the customer has the cashier and proceeds to check out. When the customer completes the checkout process, he notifies the cashier that he is finished by invoking the cashier's `done()` method. An implementation of the `Customer` class is shown in Figure 11-44.

```
import java.util.Random;

/**
 * A simple example to demonstrate the use of notifyAll() in
 * Java. This program simulates a number of customers who
 * want to check out of a store. An instance of a cashier is
 * used to control the order in which customers are served.
 */
public class Customer extends Thread {
    // Max items a customer may buy
    public static final int MAX_ITEMS = 25;
```

continued

```
private int id;                    // This customer's ID
private int numItems;              // Number of items bought
private Cashier register;          // The only register in the store

/**
 * Create a customer with the specified ID who wants to
 * go through the specified register.
 *
 * @param id this customer's ID
 * @param register the cashier who will serve this customer
 */
public Customer( int id, Cashier register ) {
    // Record state
    this.id = id;
    this.register = register;

    // Determine the number of items ( between 1 and MAX_ITEMS)
    numItems = new Random().nextInt( MAX_ITEMS ) + 1;

    // Indicate that a customer has been created
    System.out.println( "Customer " +
                        id +
                        " has " +
                        numItems +
                        " items." );
}

/**
 * This method simulates the behavior of a customer. The
 * customer requests to move to the head of the line. Once
 * there, he checks out and then leaves the line.
 */
public void run() {
    // Move to the head of the line
    register.readyToCheckOut();
    System.out.println( "Customer " + id + " is checking out" );

    // I have the cashier, so check out
    try {
        sleep( 500 );
    }
    catch ( InterruptedException e ) {}

    // That's it
    System.out.println( "Customer " +
                        id +
                        " has finished checking out" );
```

continued

```
        register.done();
    }

    /**
     * Return the ID associated with this customer.
     *
     * @return the customer's ID
     */
    public int getCustomerId() {
        return id;
    }

    /**
     * Return the number of items this customer wants to buy.
     *
     * @return the number of items purchased by the customer
     */
    public int getNumItems() {
        return numItems;
    }

} // Customer
```

[FIGURE 11-44] `Customer` class

Although the customers run as threads, the `Cashier` class is responsible for synchronizing the threads in this program. The cashier must ensure that at most one customer is checking out at a time and must determine the order in which waiting customers are served. The code required to synchronize the threads in the program is contained in the `readyToCheckOut()` and `done()` methods. The code in Figure 11-45 contains an implementation of a cashier that serves one customer at a time; however, it does not serve the customers in the order they arrive (in other words, the order in which they invoke `readyToCheckOut()`).

```
/**
 * A cashier class that shows how to use Java synchronization
 * primitives to gain access to a cashier.
 */
public class Cashier1 implements Cashier {
    private boolean busy = false;  // Is the cashier busy?

    /**
     * Invoked by a customer who is ready to check out.
     * Contains the logic required to select the one customer
```

continued

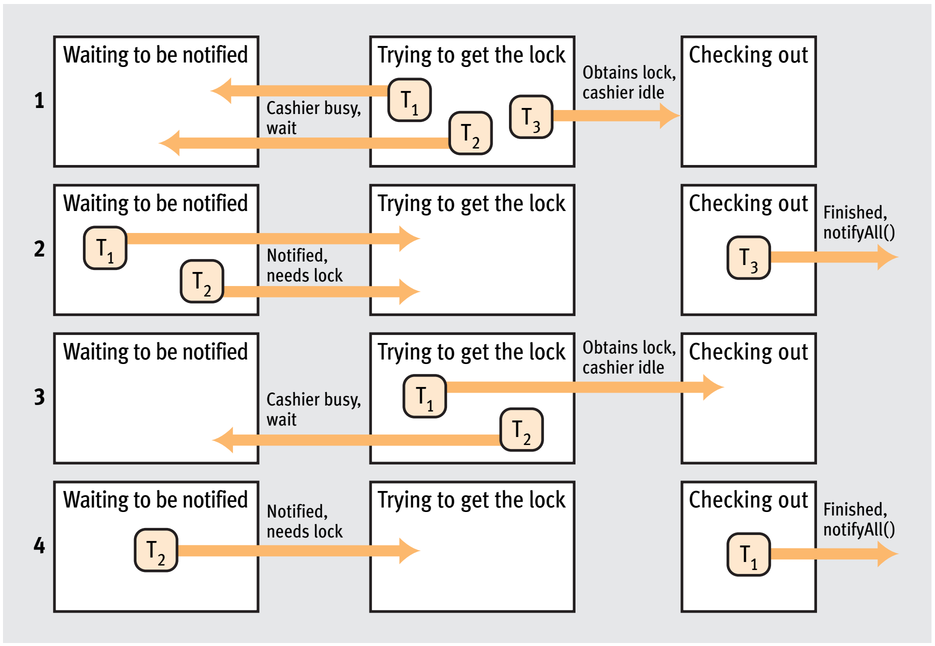
```
       * the cashier will serve. If the cashier is
       * serving another customer, the next customer waits until
       * the first one has finished with the cashier.
       */
      public synchronized void readyToCheckOut() {
          // While the cashier is busy, wait
          while ( busy ) {
              try {
                  wait();
              }
              catch (InterruptedException e ){}
          }

          // Move to the head of the line
          busy = true;
      }

      /**
       * Invoked by a customer who is finished with the
       * cashier. If customers are waiting for the cashier,
       * one is selected to move to the head of the line.
       */
      public synchronized void done() {
        if ( busy ) {
            // The cashier is no longer busy
            busy = false;

            // Let someone move to the head of the line
            notifyAll();
        }
      }

  } // Cashier1
```

[FIGURE 11-45] Simple cashier

The `Cashier` in Figure 11-45 maintains a Boolean variable, `busy`, that indicates whether a customer is currently checking out. If a customer is checking out, `busy` is set to true. If `busy` is set to false, the cashier is idle. The `readyToCheckOut()` method uses `busy` to determine if the cashier is currently serving a customer. If the cashier is busy, the customer waits until the cashier is idle. The first customer that invokes `readyToCheckOut()` determines that the cashier is idle, sets `busy` to true, and proceeds to check out. Additional customers that invoke `readyToCheckOut()` before the first customer is done must wait.

When a customer invokes `done()` to indicate he has checked out, `busy` is set to false and `notifyAll()` is invoked to signal any waiting customers that the cashier is now idle, which causes them to compete for the lock. The customer that obtains the lock resumes execution of the while loop. Because `busy` is now false, the loop terminates and the customer is given access to the cashier. The remaining customers, after obtaining the lock, find that `busy` is true and wait again. This process continues until all customers are served.

Figure 11-46 illustrates three customers working their way through the checkout process. After the threads are created, they all attempt to execute the `readyToCheckOut()` method and as a result need to obtain the lock. One thread, T_3, obtains the lock and proceeds to check out. The other two threads, T_1 and T_2, obtain the lock, determine that the cashier is busy, and invoke `wait()`. When T_3 finishes checking out, it invokes `notifyAll()` within the `done()` method, which releases T_1 and T_2 and allows them to compete for the lock again. This time, T_1 obtains the lock and checks out. T_2 again determines that the cashier is busy and waits. Once T_1 is finished, it notifies the lock, which in turn releases T_2. Now that T_2 is the only thread in the program, it obtains the lock and checks out.



[FIGURE 11-46] Using `wait()` and `notifyAll()` in the simple cashier program

In this implementation of the cashier, the program does not determine the order in which customers are served. This order is determined by the JVM based on how it assigns the ownership of the lock to the customer. The implementation of the `Cashier` interface in Figure 11-47 first serves all waiting customers that have 10 or fewer items.

```
/**
 * A cashier that gives priority access to customers who have
 * 10 items or less.
 */
public class Cashier2 implements Cashier {
    private boolean busy = false;  // Is the cashier busy?
    private int tenOrLess = 0;     // Customers with 10 or fewer items

    /**
     * Invoked by a customer who is ready to check out.
     * Contains the logic required to select the one
     * customer the cashier will serve. If the cashier is
     * serving another customer, the next customer waits until
     * the first one has finished with the cashier.
     * Customers with 10 or fewer items are allowed to move
     * ahead of other customers.
     */
    public synchronized void readyToCheckOut() {
      // Get a reference to the customer executing this code and find
      // out how many items they have
      Customer me = (Customer)Thread.currentThread();
      int items = me.getNumItems();

      // Make a note if they have 10 or fewer items
      if ( items <= 10 ) tenOrLess++;

      // As long as the cashier is busy, or someone is in line
      // with 10 or fewer items and I have more than 10, wait.
      while ( busy || tenOrLess > 0 && items > 10 )
          try {
            wait();
          }
          catch (InterruptedException e ){}

      // My chance to get out of this store!!
      busy = true;
    }

    /**
     * Invoked by a customer who is finished with the
     * cashier. If customers are waiting for the cashier,
     * those with 10 or fewer items are allowed to go
     * to the head of the line before the other customers.
     */
    public synchronized void done() {
      if ( busy ) {
          // Get a reference to the current customer
          Customer me = (Customer)Thread.currentThread();
```

continued

```
            // If they had 10 or fewer items, they are finished
            if ( me.getNumItems() <= 10 ) tenOrLess--;

            // The cashier is no longer busy
            busy = false;

            // Let someone else move to the head of the line.
            notifyAll();
        }
    }

} // Cashier2
```

[FIGURE 11-47] A cashier that gives priority to customers with 10 or fewer items

The `Cashier` class in Figure 11-47 has an additional state variable, `tenOrLess`, which keeps track of the number of waiting customers with 10 or fewer items. When a customer that has 10 or fewer items invokes `readyToCheckOut()`, the state variable `tenOrLess` is incremented by 1. When a customer with 10 or fewer items invokes `done()`, the variable is decremented by 1.

The while loop in the `readyToCheckOut()` method in Figure 11-47 uses the `busy` variable in the same way as in Figure 11-46; if the cashier is busy, the customer waits. If the `readyToCheckOut()` method in Figure 11-47 discovers that the cashier is idle, it checks to see how many items each customer has. If a customer with more than 10 items is in line waiting to check out, and another customer has fewer than 10 items, the customer with more items must wait.

Like the previous version of the cashier program, it cannot determine the exact order in which customers are served; however, all customers with 10 or fewer items are served before customers with more than 10 items. In addition to showing how a program can influence the way threads are scheduled when using `wait()` and `notifyAll()`, the program demonstrates the more general problem of **starvation**. In the cashier program of Figure 11-47, other customers are not served as long as any customer with 10 or fewer items is in line. Thus, some customers may never be able to check out in this program.

11.4 Summary

This chapter discussed how to write a program in Java that contains multiple threads of execution. Such programs can appear to do more than one thing at a time, a programming technique that is common in modern software systems. A thread in Java is represented by an instance of the `Thread` class, which contains all of the states the JVM requires to manage the execution of a thread. A program specifies the code that a thread executes by writing a class that either extends the `Thread` class or implements the `Runnable` interface. In either case, the code that the thread executes is contained in the `run()` method, which is invoked by the JVM when the thread is scheduled to execute.

The JVM determines the order in which the threads in a program are executed. The programmer can use thread priorities to give the scheduler information that helps it schedule threads, but in the end, the JVM makes the decision. Because the execution of multiple threads can be interleaved in an arbitrary number of ways, unexpected interactions between threads can result in programs that produce deadlock or incorrect results.

You can synchronize threads within a program using the synchronization primitives provided by Java. The basic tool that provides synchronization is a lock; every object in Java has an associated lock. A synchronized block can associate an object's lock with a section of code in a program. To enter a synchronized block, a thread must obtain the lock for the associated object. Because only one thread can hold a lock at a time, you are guaranteed that only one thread will execute the code in that block at a time. You can use the `wait()` and `notify()` methods within a synchronized block to suspend and resume the execution of a thread.

By placing support for threads in the JVM, Java allows you to write a multithreaded program for a platform even if the platform does not support threads. Threads and synchronization in Java are just as portable as any of the other program features we have discussed in this text. Another benefit of building support for threads and synchronization within the JVM is that you can develop special versions of the JVM to provide additional support for parallelism without changing the code. Versions of the JVM already exist that take advantage of machines with multiple processors by allowing multithreaded programs and true parallel programming.

The next chapter discusses how to use the classes in the standard Java library to build programs that interact with users through a graphical user interface.

C. A. R. HOARE

Sir Charles Antony Richard Hoare is a founding father of computer science. He studied philosophy at Oxford University, where he was introduced to the power of mathematical logic. After completing his education and serving in the British Royal Navy, he took a job as a programmer at Elliot Brothers, a small computer manufacturing firm in London.

One of his first tasks in the job was to implement a library subroutine for a new sorting technique invented by Donald Shell (commonly referred to as Shell sort). In the process of implementing this algorithm, Hoare invented a new, faster method of sorting. When he told his boss that he had developed a faster algorithm, his boss bet him six pence that he had not. After Hoare described the algorithm, which is known today as Quicksort, his boss realized he had lost the bet.

After working eight years at Elliot Brothers, Hoare became a Professor of Computing Science at the Queen's University of Belfast. He later moved back to Oxford to lead the Programming Research Group in the university's Computing Laboratory. During his tenure at Oxford he developed the CSP programming model. CSP is a theoretical model that can describe concurrent computing, and is the basis of many concurrent programming languages and libraries. Java's synchronized block is modeled after the concept of a monitor, another basic concept introduced in CSP.

Hoare received the 1980 ACM Turing Award for "his fundamental contributions to the definition and design of programming languages." In 2000 he was knighted for services to education and computer science. After retiring from teaching, he returned to industry and became a senior researcher with Microsoft Research in Cambridge.

EXERCISES

1 Look at a program that you run on a daily basis. Identify how the program might use threads.

2 Describe two reasons a thread might make a transition from the run state to the wait state.

3 Describe two reasons a thread might make a transition from the run state directly to the ready state.

4 Even if an operating system could perfectly schedule system processes so that the processor was executing user code all the time, the system would still not have 100 percent utilization. Why?

5 When using an interactive program on a computer, how can you tell when the system is busy? What happens to the performance of the program you are running?

6 Imagine you are writing a program that uses a thread to perform some time-consuming calculation. You want the program to display a message on the screen indicating that the calculation has begun. Why can't you use `isAlive()` to do this? Give a brief description of the code that you would write to perform this task.

7 How many different ways can the instruction within the `run()` method in Figure 11-17 be interleaved? How many of these interleavings results in a correct answer? What is the probability that it can produce the correct answer without some type of synchronization in the program?

8 Compile and run the program in Figure 11-17 one hundred times. How many times does it give the correct answer? How do your results compare to your analysis from Exercise 7? Try increasing the number of threads the program generates. How does this affect the program's reliability? Explain the results you observe.

9 Change the `run()` method of the program of Figure 11-17, as shown in the following code:

```
public void run() {
    int local = 0;  // Local storage

    // Add one to common
    local = common;
    local = local + 1;
    yield();
    common = local;
}
```

Repeat your experiments from Exercise 8. How does the performance of the modified program compare with the unmodified program? Explain your answer.

10 Write a Java program that prints the names and total number of all the active threads in a Java program. You might want to use the `enumerate()` method defined in the class `Thread`. After the program is working, modify it to print status information for each thread, including its priority, whether it is a daemon thread, whether the thread is alive, and so on.

11 Write a Java program that creates a number of threads, each of which prints a specific letter of the alphabet 10 times, one character per line. Run the program several times on your computing system. Is the output the same each time? Try running the same program on a different computer system several times. Is the output the same each time? Does it match the output you obtain on the first system? Rewrite your thread code so that it invokes `yield()` or `sleep()` at various points in the program. Does the output change?

12 A typical classroom setting has one professor and several students. If more than one student wants to ask a question at the same time, the professor must use a scheduling algorithm to serve the students. How would you describe this algorithm? Is it preemptive or nonpreemptive? Is it priority based?

13 In the area of road traffic, the computer science concept of deadlock is often called *gridlock*. In major cities whose roads are laid out in a grid, how does traffic become gridlocked? Describe some techniques that city planners use to prevent gridlock.

14 Complete the producer/consumer example described in the text. Write a `Producer` class that produces one integer value at a time and a `Consumer` class that consumes one integer value at a time. Connect the producer and consumer using the `MonitorBuffer` class in Figure 11-42.

15 Modify the buffer you developed for Exercise 14 to hold more than one integer value. Add a constructor to the class that allows you to specify the number of integer values the buffer can hold. Test your buffer using the `Producer` and `Consumer` classes you wrote in Exercise 14.

16 Conway's problem is defined as follows: Write a program to read a number of 80-character lines and write them to a file as 125-character lines with the following changes: A space is printed between each of the 80-character lines, and every pair of adjacent ** characters is replaced by a ^.

Write two versions of this program. The first should not use any threads, but the second should use three threads: one reads in the 80-character lines that add a space between consecutive lines, a second looks for pairs of ** characters and replaces them with a ^, and a third writes the resulting stream of characters as 125-character lines.

Place a bounded buffer between the threads in the concurrent version of the program. Design your buffer so that you can easily change its capacity. Experiment with the program to determine if the size of the buffer affects the performance.

17 Some overly energetic students learn about threads and decide to use them in a sorting program. Their idea is to create a thread that sorts numbers at the same time the main program is executing. However, the program's behavior is strange. Sometimes the numbers print in sorted order, sometimes they are partially sorted, and sometimes they are not sorted at all. What is the problem with the following program? How would you fix the problem by changing only the code in the `main()` method? Because the program sometimes prints the numbers in sorted order, you can assume that the sort routine is working correctly.

```
public class Sorter extends Thread {
    private int data[] = null;

    public Sorter( int data[] ) {
        this.data = data;
    }

    public void run() {
        // Sort the numbers (the code is correct)
        for (int i = 0; i < data.length - 1; i++)
            for ( int j = i + 1; j < data.length; j++)
                if ( data[j] < data[i] ) {
                    int tmp = data[i];
                    data[i] = data[j];
                    data[j] = tmp;
                }
    }

    public static void main( String args[] ) {
        // Generate some numbers to sort
        int data[] = new int[ 10 ];
        for (int i = 0; i < data.length; i++)
            data[i] = (int) (Math.random() * 100);

        Sorter x = new Sorter( data );
        x.start();

        for (int i = 0; i < data.length; i++)
            System.out.println( data[i] );
    }
}
```

18 Write a program to determine the accuracy of the `sleep()` program on your computing system. After your initial tests, change the program to create a number of threads that use the processor in some fashion (perhaps by executing a while loop). Does this change the accuracy of `sleep()` at all? Explain your answer.

19 Write a method named `joinAll()` that takes as a parameter an array of `Thread` references. The method returns after joining with all of the threads specified in the array. Do not make any assumptions about the order in which the threads terminate.

20 Problems with concurrent access to shared resources can occur when a team of programmers is working together to write a program. If two team members make changes to the same piece of code at the same time, one set of changes might be lost. CVS is a version control system that tries to eliminate this problem. Look up the documentation for CVS using an Internet search engine, and discuss how CVS resolves the concurrent access problem.

21 Rewrite your program from Exercise 8 using locks and shared variables so that the order in which the threads execute is deterministic (in other words, you specify the order of execution).

22 What output is generated by the following program?

```
public class Foobar extends Thread {

    private static synchronized void method() {
        System.out.println( "—>" );

        try {
            Thread.sleep( 1000 );
        }
        catch ( InterruptedException ex ) {}

        System.out.println( "<—" );
    }

    public void run() {
        method();
    }

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

23 What output is generated by the following program?

```
public class Class2 extends Thread {
    private static class Class1 {
        private int n;
        private List<Integer> a;

        public Class1( int x ) {
            a = new ArrayList<Integer>();
            n = x;
        }

        public synchronized void method1( int x ) {
            a.add( x );
            n = n - 1;

            if ( n > 0 )
                try {
                    wait();
                }
                catch ( InterruptedException e ) {}
            else {
                Collections.sort( a );
                notifyAll();
            }
        }

        public synchronized void method2( int y ) {
            boolean x = y % 2 != 0; // Is y odd?
            boolean p = false;

            while ( !p ) {
                int i = a.get( 0 );
                p = i == y;

                if ( p && x ) {
                    a.remove( 0 );
                    a.add( i );
                    notifyAll();
                    x = false;
                    p = false;
                }
```

continued

```
                if ( !p ) {
                    try {
                        wait();
                    }
                    catch ( InterruptedException e ) {}
                }
            }
        }

        public synchronized void method3() {
            a.remove( 0 );
            notifyAll();
        }
    } // Class1

    private Class1 one;
    private int a;

    public Class2( Class1 one, int a ) {
        this.one = one;
        this.a = a;
    }

    public void run() {
        one.method1( a );
        one.method2( a );
        System.out.println( a );
        one.method3();
    }

    public static void main( String args[] ) {
        Class1 one = new Class1( 10 );
        for ( int i = 0; i < 10; i++ )
            new Class2( one, i ).start();
    }
} // Class2
```

24 A semaphore is a classic synchronization tool in many operating systems. Look up the definition of *semaphore* and then write a class named `Semaphore` that provides a Java-based implementation of a semaphore. Test your implementation by rewriting the producer/consumer code in this chapter so that it uses semaphores instead of Java locks.

25 The cashier program described in Section 11.3.3 does not maintain a line. Normally, customers are served by a cashier on a first-come, first-served basis (in other words, the customer at the front of the line checks out next). The code given in the book selects the next customer at random. Modify this code so that customers are served in the order they arrive at the cashier.

CHALLENGE WORK EXERCISES

1 Some people consider Java's synchronization facilities to be primitive. This led to the development of the `java.util.concurrent` packages that are now included in the standard Java API. Study the locking facilities provided by the `java.util.concurrent.locks` package. How do these facilities differ from the basic synchronization facilities provided by the language? Rewrite the `Cashier` example in Section 11.3.3 to use the classes in the `java.util.concurrent.locks` package.

2 How do the threading and synchronization facilities of the ADA programming language compare to those provided by Java? Convert the programs that illustrate the basic threading concepts in this chapter to ADA. Give strengths and weaknesses of the facilities provided by each language.

3 One of the most famous examples of a problem that can occur with concurrent programs is the dining philosophers problem. Imagine a group of philosophers around a table waiting for dinner to be served. The table has a plate for each philosopher and a single fork between each plate. Before eating, a philosopher must pick up the forks on both sides of his plate. Only one philosopher can hold forks at a time, and no one else can use the forks until they are placed back on the table.

The philosophers must develop a procedure so they can eat. One possibility is for each philosopher to attempt to pick up the fork on the left side of his plate. If the left fork is being used by the philosopher on the left, he waits until the fork is placed back on the table, and then picks it up. After he has the left fork, he repeats the same process to obtain the right fork. Once a philosopher has both forks, he eats and then places both forks on the table when he is finished.

Does this procedure always work? If not, give an example of when it will fail. Develop a modified procedure that works. After confirming that it works, write a Java program to simulate the solution.

4 A barbershop consists of a waiting room with *N* chairs and a barber chair. If there are no customers, the barber goes to sleep. If a customer enters and all the chairs are occupied, the customer leaves the shop without getting a haircut. If the barber is busy but chairs are available, the customer waits in a chair for a haircut. If the barber is asleep, the customer wakes him up and gets a haircut. Write a Java program to simulate this barbershop.

5 Consider a system that consists of three smoker threads and one agent thread. Each smoker continuously rolls a cigarette and smokes it. To do so, the smoker needs three ingredients: tobacco, paper, and matches. One of the smoker threads has paper, another has tobacco, and the third has matches. The agent has an infinite supply of all three materials. The agent places two of the ingredients on the table. The smoker who has the remaining ingredient then makes and smokes a cigarette, signaling the agent when it is done. The agent then puts another two of the three ingredients on the table, and the cycle repeats. Write a Java program to simulate the smoker and agent threads.