Objectives

In this chapter you’ll learn:

■ What threads are and why they’re useful.
■ How threads enable you to manage concurrent activities.
■ The life cycle of a thread.
■ To create and execute Runnable s.
■ Thread synchronization.
■ What producer/consumer relationships are and how they’re implemented with multithreading.
■ To enable multiple threads to update Swing GUI components in a thread-safe manner.
Chapter 26  Multithreading

26.1 Introduction

It would be nice if we could focus our attention on performing only one action at a time and performing it well, but that’s usually difficult to do. The human body performs a great variety of operations in parallel—or, as we’ll say throughout this chapter, concurrently. Respiration, blood circulation, digestion, thinking and walking, for example, can occur concurrently, as can all the senses—sight, touch, smell, taste and hearing.

Computers, too, can perform operations concurrently. It’s common for personal computers to compile a program, send a file to a printer and receive electronic mail messages over a network concurrently. Only computers that have multiple processors can truly execute multiple instructions concurrently. Operating systems on single-processor computers create the illusion of concurrent execution by rapidly switching between activities, but on such computers only a single instruction can execute at once. Today’s multicore computers have multiple processors that enable computers to perform tasks truly concurrently. Multicore smartphones are starting to appear.

Historically, concurrency has been implemented with operating system primitives available only to experienced systems programmers. The Ada programming language—developed by the United States Department of Defense—made concurrency primitives widely available to defense contractors building military command-and-control systems. However, Ada has not been widely used in academia and industry.

Java Concurrency

Java makes concurrency available to you through the language and APIs. Java programs can have multiple threads of execution, where each thread has its own method-call stack and program counter, allowing it to execute concurrently with other threads while sharing with them application-wide resources such as memory. This capability is called multithreading.
We’ll discuss many applications of concurrent programming. For example, when downloading a large file (e.g., an image, an audio clip or a video clip) over the Internet, the user may not want to wait until the entire clip downloads before starting the playback. To solve this problem, multiple threads can be used—one to download the clip, and another to play it. These activities proceed concurrently. To avoid choppy playback, the threads are synchronized (that is, their actions are coordinated) so that the player thread doesn’t begin until there’s a sufficient amount of the clip in memory to keep the player thread busy. The Java Virtual Machine (JVM) creates threads to run programs and threads to perform housekeeping tasks such as garbage collection.

Writing multithreaded programs can be tricky. Although the human mind can perform functions concurrently, people find it difficult to jump between parallel trains of thought. To see why multithreaded programs can be difficult to write and understand, try the following experiment: Open three books to page 1, and try reading the books concurrently. Read a few words from the first book, then a few from the second, then a few from the third, then loop back and read the next few words from the first book, and so on. After this experiment, you’ll appreciate many of the challenges of multithreading—switching between the books, reading briefly, remembering your place in each book, moving the book you’re reading closer so that you can see it and pushing the books you’re not reading aside—and, amid all this chaos, trying to comprehend the content of the books!

Programming concurrent applications is difficult and error prone. If you must use synchronization in a program, you should follow some simple guidelines. Use existing classes from the Concurrency APIs (such as the ArrayBlockingQueue class we discuss in Section 26.6) that manage synchronization for you. These classes are written by experts, have been thoroughly tested and debugged, operate efficiently and help you avoid common traps and pitfalls.

If you need even more complex capabilities, use interfaces Lock and Condition that we introduce in Section 26.9. These interfaces should be used only by advanced programmers who are familiar with concurrent programming’s common traps and pitfalls. We explain these topics in this chapter for several reasons:

- They provide a solid basis for understanding how concurrent applications synchronize access to shared memory.
- The concepts are important to understand, even if an application does not use these tools explicitly.
Chapter 26  Multithreading

• By showing you the complexity involved in using these low-level features, we hope to impress upon you the importance of using prebuilt concurrency capabilities whenever possible.

Section 26.10 provides an overview of Java’s pre-built concurrent collections.

26.2 Thread States: Life Cycle of a Thread

At any time, a thread is said to be in one of several thread states—illustrated in the UML state diagram in Fig. 26.1. Several of the terms in the diagram are defined in later sections. We include this discussion to help you understand what’s going on “under the hood” in a Java multithreaded environment. Java hides most of this detail from you, greatly simplifying the task of developing multithreaded applications.

![Thread life-cycle UML state diagram.](image)

Fig. 26.1  |  Thread life-cycle UML state diagram.

New and Runnable States

A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread, which places it in the runnable state. A thread in the runnable state is considered to be executing its task.

Waiting State

Sometimes a runnable thread transitions to the waiting state while it waits for another thread to perform a task. A waiting thread transitions back to the runnable state only when another thread notifies it to continue executing.

Timed Waiting State

A runnable thread can enter the timed waiting state for a specified interval of time. It transitions back to the runnable state when that time interval expires or when the event it’s waiting for occurs. Timed waiting and waiting threads cannot use a processor, even if one
is available. A runnable thread can transition to the timed waiting state if it provides an optional wait interval when it’s waiting for another thread to perform a task. Such a thread returns to the runnable state when it’s notified by another thread or when the timed interval expires—whichever comes first. Another way to place a thread in the timed waiting state is to put a runnable thread to sleep. A sleeping thread remains in the timed waiting state for a designated period of time (called a sleep interval), after which it returns to the runnable state. Threads sleep when they momentarily do not have work to perform. For example, a word processor may contain a thread that periodically backs up (i.e., writes a copy of) the current document to disk for recovery purposes. If the thread did not sleep between successive backups, it would require a loop in which it continually tested whether it should write a copy of the document to disk. This loop would consume processor time without performing productive work, thus reducing system performance. In this case, it’s more efficient for the thread to specify a sleep interval (equal to the period between successive backups) and enter the timed waiting state. This thread is returned to the runnable state when its sleep interval expires, at which point it writes a copy of the document to disk and reenters the timed waiting state.

Blocked State
A runnable thread transitions to the blocked state when it attempts to perform a task that cannot be completed immediately and it must temporarily wait until that task completes. For example, when a thread issues an input/output request, the operating system blocks the thread from executing until that I/O request completes—at that point, the blocked thread transitions to the runnable state, so it can resume execution. A blocked thread cannot use a processor, even if one is available.

Terminated State
A runnable thread enters the terminated state (sometimes called the dead state) when it successfully completes its task or otherwise terminates (perhaps due to an error). In the UML state diagram of Fig. 26.1, the terminated state is followed by the UML final state (the bull’s-eye symbol) to indicate the end of the state transitions.

Operating-System View of the Runnable State
At the operating system level, Java’s runnable state typically encompasses two separate states (Fig. 26.2). The operating system hides these states from the Java Virtual Machine (JVM), which sees only the runnable state. When a thread first transitions to the runnable state from the new state, it’s in the ready state. A ready thread enters the running state (i.e., begins executing) when the operating system assigns it to a processor—also known as dispatching the thread. In most operating systems, each thread is given a small amount of processor time—called a quantum or timeslice—with which to perform its task. Deciding how large the quantum should be is a key topic in operating systems courses. When its quantum expires, the thread returns to the ready state, and the operating system assigns another thread to the processor. Transitions between the ready and running states are handled solely by the operating system. The JVM does not “see” the transitions—it simply views the thread as being runnable and leaves it up to the operating system to transition the thread between ready and running. The process that an operating system uses to determine which thread to dispatch is called thread scheduling and is dependent on thread priorities.
Chapter 26  Multithreading

Thread Priorities and Thread Scheduling

Every Java thread has a thread priority that helps determine the order in which threads are scheduled. Each new thread inherits the priority of the thread that created it. Informally, higher-priority threads are more important to a program and should be allocated processor time before lower-priority threads. Nevertheless, thread priorities cannot guarantee the order in which threads execute.

It’s recommended that you do not explicitly create and use Thread objects to implement concurrency, but rather use the Executor interface (which is described in Section 26.3). The Thread class does contain some useful static methods, which you will use later in the chapter.

Most operating systems support timeslicing, which enables threads of equal priority to share a processor. Without timeslicing, each thread in a set of equal-priority threads runs to completion (unless it leaves the runnable state and enters the waiting or timed waiting state, or gets interrupted by a higher-priority thread) before other threads of equal priority get a chance to execute. With timeslicing, even if a thread has not finished executing when its quantum expires, the processor is taken away from the thread and given to the next thread of equal priority, if one is available.

An operating system’s thread scheduler determines which thread runs next. One simple thread-scheduler implementation keeps the highest-priority thread running at all times and, if there’s more than one highest-priority thread, ensures that all such threads execute for a quantum each in round-robin fashion. This process continues until all threads run to completion.

When a higher-priority thread enters the ready state, the operating system generally preempts the currently running thread (an operation known as preemptive scheduling). Depending on the operating system, higher-priority threads could postpone—possibly indefinitely—the execution of lower-priority threads. Such indefinite postponement is sometimes referred to more colorfully as starvation. Operating systems employ a technique called aging to prevent starvation—as a thread waits in the ready state, the operating system gradually increases the thread’s priority, thus ensuring that the thread will eventually run.

Java provides higher-level concurrency utilities to hide much of this complexity and make multithreaded programming less error prone. Thread priorities are used behind the scenes to interact with the operating system, but most programmers who use Java multithreading will not be concerned with setting and adjusting thread priorities.

Portability Tip 26.1

Thread scheduling is platform dependent—the behavior of a multithreaded program could vary across different Java implementations.
26.3 Creating and Executing Threads with Executor Framework

This section demonstrates how to perform concurrent tasks in an application by using Executors and Runnable objects.

Creating Concurrent Tasks with the Runnable Interface
You implement the Runnable interface (of package java.lang) to specify a task that can execute concurrently with other tasks. The Runnable interface declares the single method run, which contains the code that defines the task that a Runnable object should perform.

Executing Runnable Objects with an Executor
To allow a Runnable to perform its task, you must execute it. An Executor object executes Runnables. An Executor does this by creating and managing a group of threads called a thread pool. When an Executor begins executing a Runnable, the Executor calls the Runnable object’s run method, which executes in the new thread.

The Executor interface declares a single method named execute which accepts a Runnable as an argument. The Executor assigns every Runnable passed to its execute method to one of the available threads in the thread pool. If there are no available threads, the Executor creates a new thread or waits for a thread to become available and assigns that thread the Runnable that was passed to method execute.

Using an Executor has many advantages over creating threads yourself. Executors can reuse existing threads to eliminate the overhead of creating a new thread for each task and can improve performance by optimizing the number of threads to ensure that the processor stays busy, without creating so many threads that the application runs out of resources.

Software Engineering Observation 26.1
Though it’s possible to create threads explicitly, it’s recommended that you use the Executor interface to manage the execution of Runnable objects.

Using Class Executors to Obtain an ExecutorService
The ExecutorService interface (of package java.util.concurrent) extends Executor and declares various methods for managing the life cycle of an Executor. An object that implements the ExecutorService interface can be created using static methods declared in class Executors (of package java.util.concurrent). We use interface ExecutorService and a method of class Executors in our example, which executes three tasks.

Implementing the Runnable Interface
Class PrintTask (Fig. 26.3) implements Runnable (line 5), so that multiple PrintTasks can execute concurrently. Variable sleepTime (line 7) stores a random integer value from 0 to 5 seconds created in the PrintTask constructor (line 17). Each thread running a PrintTask sleeps for the amount of time specified by sleepTime, then outputs its task’s name and a message indicating that it’s done sleeping.

A PrintTask executes when a thread calls the PrintTask’s run method. Lines 25–26 display a message indicating the name of the currently executing task and that the task is going to sleep for sleepTime milliseconds. Line 27 invokes static method sleep of class Thread to place the thread in the timed waiting state for the specified amount of time. At this point, the thread loses the processor, and the system allows another thread to execute.
When the thread awakens, it reenters the **runnable** state. When the PrintTask is assigned to a processor again, line 36 outputs a message indicating that the task is done sleeping, then method run terminates. The catch at lines 29–33 is required because method sleep might throw a **checked** exception of type **InterruptedException** if a sleeping thread’s interrupt method is called.

**Using the ExecutorService to Manage Threads that Execute PrintTasks**

Figure 26.4 uses an ExecutorService object to manage threads that execute PrintTasks (as defined in Fig. 26.3). Lines 11–13 create and name three PrintTask objects to execute. Line 18 uses Executors method **newCachedThreadPool** to obtain an ExecutorService that’s capable of creating new threads as they’re needed by the application. These threads are used by ExecutorService (threadExecutor) to execute the Runnables.
26.3 Creating and Executing Threads with Executor Framework

```java
public class TaskExecutor {
    public static void main(String[] args) {
        // create and name each runnable
        PrintTask task1 = new PrintTask("task1");
        PrintTask task2 = new PrintTask("task2");
        PrintTask task3 = new PrintTask("task3");

        System.out.println("Starting Executor");

        // create ExecutorService to manage threads
        ExecutorService threadExecutor = Executors.newCachedThreadPool();
        threadExecutor.execute(task1); // start task1
        threadExecutor.execute(task2); // start task2
        threadExecutor.execute(task3); // start task3

        // shut down worker threads when their tasks complete
        threadExecutor.shutdown();

        System.out.println("Tasks started, main ends.\n");
    }
}
```

Fig. 26.4  Using an ExecutorService to execute Runnable objects.
Lines 21–23 each invoke the `ExecutorService`'s `execute` method, which executes the `Runnable` passed to it as an argument (in this case a `PrintTask`) some time in the future. The specified task may execute in one of the threads in the `ExecutorService`'s thread pool, in a new thread created to execute it, or in the thread that called the `execute` method—the `ExecutorService` manages these details. Method `execute` returns immediately from each invocation—the program does not wait for each `PrintTask` to finish. Line 26 calls `ExecutorService` method `shutdown`, which notifies the `ExecutorService` to stop accepting new tasks, but continues executing tasks that have already been submitted. Once all of the previously submitted `Runnables` have completed, the `threadExecutor` terminates. Line 28 outputs a message indicating that the tasks were started and the `main` thread is finishing its execution.

The code in `main` executes in the `main thread`, a thread created by the JVM. The code in the run method of `PrintTask` (lines 21–37 of Fig. 26.3) executes whenever the `Executor` starts each `PrintTask`—again, this is sometime after they’re passed to the `ExecutorService`'s `execute` method (Fig. 26.4, lines 21–23). When `main` terminates, the program itself continues running because there are still tasks that must finish executing. The program will not terminate until these tasks complete.

The sample outputs show each task’s name and sleep time as the thread goes to sleep. The thread with the shortest sleep time normally awakens first, indicates that it’s done sleeping and terminates. In Section 26.8, we discuss multithreading issues that could prevent the thread with the shortest sleep time from awakening first. In the first output, the `main` thread terminates before any of the `PrintTasks` output their names and sleep times. This shows that the `main` thread runs to completion before any of the `PrintTasks` get a chance to run. In the second output, all of the `PrintTasks` output their names and sleep times before the `main` thread terminates. This shows that the `PrintTasks` started executing before the `main` thread terminated. Also, notice in the second example output, `task3` goes to sleep before `task2` last, even though we passed `task2` to the `ExecutorService`'s `execute` method before `task3`. This illustrates the fact that we cannot predict the order in which the tasks will start executing, even if we know the order in which they were created and started.

### 26.4 Thread Synchronization

When multiple threads share an object and it’s modified by one or more of them, indeterminate results may occur (as we’ll see in the examples) unless access to the shared object is managed properly. If one thread is in the process of updating a shared object and another thread also tries to update it, it’s unclear which thread’s update takes effect. When this happens, the program’s behavior cannot be trusted—sometimes the program will produce the correct results, and sometimes it won’t. In either case, there’ll be no indication that the shared object was manipulated incorrectly.

The problem can be solved by giving only one thread at a time exclusive access to code that manipulates the shared object. During that time, other threads desiring to manipulate the object are kept waiting. When the thread with exclusive access to the object finishes manipulating it, one of the threads that was waiting is allowed to proceed. This process, called thread synchronization, coordinates access to shared data by multiple concurrent threads. By synchronizing threads in this manner, you can ensure that each thread accessing a shared object excludes all other threads from doing so simultaneously—this is called mutual exclusion.
26.4 Thread Synchronization

Monitors

A common way to perform synchronization is to use Java’s built-in monitors. Every object has a monitor and a monitor lock (or intrinsic lock). The monitor ensures that its object’s monitor lock is held by a maximum of only one thread at any time. Monitors and monitor locks can thus be used to enforce mutual exclusion. If an operation requires the executing thread to hold a lock while the operation is performed, a thread must acquire the lock before proceeding with the operation. Other threads attempting to perform an operation that requires the same lock will be blocked until the first thread releases the lock, at which point the blocked threads may attempt to acquire the lock and proceed with the operation.

To specify that a thread must hold a monitor lock to execute a block of code, the code should be placed in a synchronized statement. Such code is said to be guarded by the monitor lock; a thread must acquire the lock to execute the guarded statements. The synchronized statements are declared using the synchronized keyword:

```
synchronized (object) {
    statements
} // end synchronized statement
```

where object is the object whose monitor lock will be acquired; object is normally this if it’s the object in which the synchronized statement appears. If several synchronized statements are trying to execute on an object at the same time, only one of them may be active on the object—all the other threads attempting to enter a synchronized statement on the same object are placed in the blocked state.

When a synchronized statement finishes executing, the object’s monitor lock is released and one of the blocked threads attempting to enter a synchronized statement can be allowed to acquire the lock to proceed. Java also allows synchronized methods. Before executing, a non-static synchronized method must acquire the lock on the object that’s used to call the method. Similarly, a static synchronized method must acquire the lock on the class that’s used to call the method.

26.4.1 Unsynchronized Data Sharing

First, we illustrate the dangers of sharing an object across threads without proper synchronization. In this example, two Runnable objects maintain references to a single integer array. Each Runnable writes three values to the array, then terminates. This may seem harmless, but we’ll see that it can result in errors if the array is manipulated without synchronization.

Class SimpleArray

A SimpleArray object (Fig. 26.5) will be shared across multiple threads. SimpleArray will enable those threads to place int values into array (declared at line 8). Line 9 initializes variable writeIndex, which will be used to determine the array element that should be written to next. The constructor (lines 13–16) creates an integer array of the desired size.

Method add (lines 19–40) allows new values to be inserted at the end of the array. Line 21 stores the current writeIndex value. Line 26 puts the thread that invokes add to sleep for a random interval from 0 to 499 milliseconds. This is done to make the problems associated with unsynchronized access to shared data more obvious. After the thread is done...
sleeping, line 34 inserts the value passed to `add` into the array at the element specified by `position`. Lines 35–36 output a message indicating the executing thread’s name, the value that was inserted in the array and where it was inserted. The expression `Thread.currentThread().getName()` (line 36) first obtains a reference to the currently executing Thread,
then uses that Thread’s getName method to obtain its name. Line 38 increments writeIndex so that the next call to add will insert a value in the array’s next element. Lines 43–46 override method toString to create a String representation of the array’s contents.

**Class ArrayWriter**
Class ArrayWriter (Fig. 26.6) implements the interface Runnable to define a task for inserting values in a SimpleArray object. The constructor (lines 10–14) takes two arguments—an integer value, which is the first value this task will insert in the SimpleArray object, and a reference to the SimpleArray object. Line 20 invokes method add on the SimpleArray object. The task completes after three consecutive integers beginning with startValue are added to the SimpleArray object.

```java
// Fig. 26.6: ArrayWriter.java
// Adds integers to an array shared with other Runnables
import java.lang.Runnable;

public class ArrayWriter implements Runnable {
    private final SimpleArray sharedSimpleArray;
    private final int startValue;

    public ArrayWriter( int value, SimpleArray array ) {
        startValue = value;
        sharedSimpleArray = array;
    } // end constructor

    public void run() {
        for ( int i = startValue; i < startValue + 3; i++ ) {
            sharedSimpleArray.add( i ); // add an element to the shared array
        } // end for
    } // end method run
}
// end class ArrayWriter
```

**Fig. 26.6** | Adds integers to an array shared with other Runnables.

**Class SharedArrayTest**
Class SharedArrayTest (Fig. 26.7) executes two ArrayWriter tasks that add values to a single SimpleArray object. Line 12 constructs a six-element SimpleArray object. Lines 15–16 create two new ArrayWriter tasks, one that places the values 1–3 in the SimpleArray object, and one that places the values 11–13. Lines 19–21 create an ExecutorService and execute the two ArrayWriters. Line 23 invokes the ExecutorService’s shutdown method to prevent additional tasks from starting and to enable the application to terminate when the currently executing tasks complete execution.

Recall that ExecutorService method shutdown returns immediately. Thus any code that appears after the call to ExecutorService method shutdown in line 23 will continue executing as long as the main thread is still assigned to a processor. We’d like to output the SimpleArray object to show you the results after the threads complete their tasks. So, we
need the program to wait for the threads to complete before main outputs the SimpleArray object’s contents. Interface ExecutorService provides the **awaitTermination** method for this purpose. This method returns control to its caller either when all tasks executing in the ExecutorService complete or when the specified timeout elapses. If all tasks are completed before **awaitTermination** times out, this method returns true; otherwise it returns false. The two arguments to **awaitTermination** represent a timeout value and a unit of measure specified with a constant from class TimeUnit (in this case, TimeUnit.MINUTES).

```java
// Fig 26.7: SharedArrayTest.java
// Executes two Runnables to add elements to a shared SimpleArray.
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;

public class SharedArrayTest {  
    public static void main( String[] arg )  
    {  
        // construct the shared object
        SimpleArray sharedSimpleArray = new SimpleArray( 6 );

        // create two tasks to write to the shared SimpleArray
        ArrayWriter writer1 = new ArrayWriter( 1, sharedSimpleArray );
        ArrayWriter writer2 = new ArrayWriter( 11, sharedSimpleArray );

        // execute the tasks with an ExecutorService
        ExecutorService executor = Executors.newCachedThreadPool();
        executor.execute( writer1 );
        executor.execute( writer2 );

        executor.shutdown();

        try  
        {  
            // wait 1 minute for both writers to finish executing
            boolean tasksEnded = executor.awaitTermination( 1, TimeUnit.MINUTES );

            if ( tasksEnded )
                System.out.println( sharedSimpleArray ); // print contents
            else
                System.out.println( "Timed out while waiting for tasks to finish." );
        } // end try
        catch ( InterruptedException ex )
        {
            System.out.println("Interrupted while waiting for tasks to finish.");
        } // end catch
    } // end main
} // end class SharedArrayTest

Fig. 26.7 | Executes two Runnables to insert values in a shared array. (Part 1 of 2.)
In this example, if both tasks complete before `awaitTermination` times out, line 32 displays the `SimpleArray` object’s contents. Otherwise, lines 34–35 print a message indicating that the tasks did not finish executing before `awaitTermination` timed out.

The output in Fig. 26.7 demonstrates the problems (highlighted in the output) that can be caused by failure to synchronize access to shared data. The value 1 was written to element 0, then overwritten later by the value 11. Also, when `writeIndex` was incremented to 3, nothing was written to that element, as indicated by the 0 in that element of the printed array.

Recall that we added calls to `Thread` method `sleep` between operations on the shared data to emphasize the unpredictability of thread scheduling and increase the likelihood of producing erroneous output. Even if these operations were allowed to proceed at their normal pace, you could still see errors in the program’s output. However, modern processors can handle the simple operations of the `SimpleArray` method add so quickly that you might not see the errors caused by the two threads executing this method concurrently, even if you tested the program dozens of times. One of the challenges of multithreaded programming is spotting the errors—they may occur so infrequently that a broken program does not produce incorrect results during testing, creating the illusion that the program is correct.

### 26.4.2 Synchronized Data Sharing—Making Operations Atomic

The output errors of Fig. 26.7 can be attributed to the fact that the shared object, `SimpleArray`, is not thread safe—`SimpleArray` is susceptible to errors if it’s accessed concurrently by multiple threads. The problem lies in method `add`, which stores the value of `writeIndex`, places a new value in that element, then increments `writeIndex`. Such a method would present no problem in a single-threaded program. However, if one thread obtains the value of `writeIndex`, there’s no guarantee that another thread cannot come along and increment `writeIndex` before the first thread has had a chance to place a value in the array. If this happens, the first thread will be writing to the array based on a stale value of `writeIndex`—a value that’s no longer valid. Another possibility is that one thread might obtain the value of `writeIndex` after another thread adds an element to the array but before `writeIndex` is incremented. In this case, too, the first thread would write to the array based on an invalid value for `writeIndex`.

---

**Fig. 26.7** | Executes two `Runnable` s to insert values in a shared array. (Part 2 of 2.)
SimpleArray is not thread safe because it allows any number of threads to read and modify shared data concurrently, which can cause errors. To make SimpleArray thread safe, we must ensure that no two threads can access it at the same time. We also must ensure that while one thread is in the process of storing writeIndex, adding a value to the array, and incrementing writeIndex, no other thread may read or change the value of writeIndex or modify the contents of the array at any point during these three operations. In other words, we want these three operations—storing writeIndex, writing to the array, incrementing writeIndex—to be an atomic operation, which cannot be divided into smaller suboperations. We can simulate atomicity by ensuring that only one thread carries out the three operations at a time. Any other threads that need to perform the operation must wait until the first thread has finished the add operation in its entirety.

Atomicity can be achieved using the synchronized keyword. By placing our three suboperations in a synchronized statement or synchronized method, we allow only one thread at a time to acquire the lock and perform the operations. When that thread has completed all of the operations in the synchronized block and releases the lock, another thread may acquire the lock and begin executing the operations. This ensures that a thread executing the operations will see the actual values of the shared data and that these values will not change unexpectedly in the middle of the operations as a result of another thread’s modifying them.

Software Engineering Observation 26.2
Place all accesses to mutable data that may be shared by multiple threads inside synchronized statements or synchronized methods that synchronize on the same lock. When performing multiple operations on shared data, hold the lock for the entirety of the operation to ensure that the operation is effectively atomic.

Class SimpleArray with Synchronization
Figure 26.8 displays class SimpleArray with the proper synchronization. Notice that it's identical to the SimpleArray class of Fig. 26.5, except that add is now a synchronized method (line 20). So, only one thread at a time can execute this method. We reuse classes ArrayWriter (Fig. 26.6) and SharedArrayTest (Fig. 26.7) from the previous example.

```java
// Fig. 26.8: SimpleArray.java
// Class that manages an integer array to be shared by multiple threads with synchronization.
import java.util.Arrays;
import java.util.Random;

public class SimpleArray
{
  private final int[] array; // the shared integer array
  private int writeIndex = 0; // index of next element to be written
  private final static Random generator = new Random();

  // Class that manages an integer array to be shared by multiple threads with synchronization. (Part 1 of 2.)
```
26.4 Thread Synchronization

```java
// construct a SimpleArray of a given size
public SimpleArray( int size )
{
    array = new int[ size ];
} // end constructor

// add a value to the shared array
public synchronized void add( int value )
{
    int position = writeIndex; // store the write index
    try {
        // put thread to sleep for 0-499 milliseconds
        Thread.sleep( generator.nextInt( 500 ) );
    } // end try
    catch ( InterruptedException ex )
    {
        ex.printStackTrace();
    } // end catch
    // put value in the appropriate element
    array[ position ] = value;
    System.out.printf( "%s wrote %2d to element %d.\n", Thread.currentThread().getName(), value, position );
    ++writeIndex; // increment index of element to be written next
    System.out.printf( "Next write index: %d\n", writeIndex );
} // end method add

// used for outputting the contents of the shared integer array
public String toString()
{
    return "\nContents of SimpleArray:\n" + Arrays.toString( array );
} // end method toString
} // end class SimpleArray
```

Fig. 26.8 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 2 of 2.)
Line 20 declares method as synchronized, making all of the operations in this method behave as a single, atomic operation. Line 22 performs the first suboperation—storing the value of writeIndex. Line 35 defines the second suboperation, writing an element to the element at the index position. Line 39 increments writeIndex. When the method finishes executing at line 41, the executing thread implicitly releases the SimpleArray lock, making it possible for another thread to begin executing the add method.

In the synchronized add method, we print messages to the console indicating the progress of threads as they execute this method, in addition to performing the actual operations required to insert a value in the array. We do this so that the messages will be printed in the correct order, allowing us to see whether the method is properly synchronized by comparing these outputs with those of the previous, unsynchronized example. We continue to output messages from synchronized blocks in later examples for demonstration purposes only; typically, however, I/O should not be performed in synchronized blocks, because it’s important to minimize the amount of time that an object is “locked.” Also, line 27 in this example calls Thread method sleep to emphasize the unpredictability of thread scheduling. You should never call sleep while holding a lock in a real application.

Another note on thread safety: We’ve said that it’s necessary to synchronize access to all data that may be shared across multiple threads. Actually, this synchronization is necessary only for mutable data, or data that may change in its lifetime. If the shared data will not change in a multithreaded program, then it’s not possible for a thread to see old or incorrect values as a result of another thread’s manipulating that data.

When you share immutable data across threads, declare the corresponding data fields final to indicate that the values of the variables will not change after they’re initialized. This prevents accidental modification of the shared data later in a program, which could compromise thread safety. Labeling object references as final indicates that the reference will not change, but it does not guarantee that the object itself is immutable—this depends entirely on the object’s properties. However, it’s still good practice to mark references that will not change as final, as doing so forces the object’s constructor to be atomic—the object will be fully constructed with all its fields initialized before the program accesses it.

26.5 Producer/Consumer Relationship without Synchronization

In a producer/consumer relationship, the producer portion of an application generates data and stores it in a shared object, and the consumer portion of the application reads data
from the shared object. The producer/consumer relationship separates the task of identifying work to be done from the tasks involved in actually carrying out the work. One example of a common producer/consumer relationship is print spooling. Although a printer might not be available when you want to print from an application (i.e., the producer), you can still “complete” the print task, as the data is temporarily placed on disk until the printer becomes available. Similarly, when the printer (i.e., a consumer) is available, it doesn’t have to wait until a current user wants to print. The spooled print jobs can be printed as soon as the printer becomes available. Another example of the producer/consumer relationship is an application that copies data onto DVDs by placing data in a fixed-size buffer, which is emptied as the DVD drive “burns” the data onto the DVD.

In a multithreaded producer/consumer relationship, a producer thread generates data and places it in a shared object called a buffer. A consumer thread reads data from the buffer. This relationship requires synchronization to ensure that values are produced and consumed properly. All operations on mutable data that’s shared by multiple threads (e.g., the data in the buffer) must be guarded with a lock to prevent corruption, as discussed in Section 26.4. Operations on the buffer data shared by a producer and consumer thread are also state dependent—the operations should proceed only if the buffer is in the correct state. If the buffer is in a not-full state, the producer may produce; if the buffer is in a not-empty state, the consumer may consume. All operations that access the buffer must use synchronization to ensure that data is written to the buffer or read from the buffer only if the buffer is in the proper state. If the producer attempting to put the next data into the buffer determines that it’s full, the producer thread must wait until there’s space to write a new value. If a consumer thread finds the buffer empty or finds that the previous data has already been read, the consumer must also wait for new data to become available.

Consider how logic errors can arise if we do not synchronize access among multiple threads manipulating shared data. Our next example (Fig. 26.9–Fig. 26.13) implements a producer/consumer relationship without the proper synchronization. A producer thread writes the numbers 1 through 10 into a shared buffer—a single memory location shared between two threads (a single int variable called buffer in line 6 of Fig. 26.12 in this example). The consumer thread reads this data from the shared buffer and displays the data. The program’s output shows the values that the producer writes (produces) into the shared buffer and the values that the consumer reads (consumes) from the shared buffer.

Each value the producer thread writes to the shared buffer must be consumed exactly once by the consumer thread. However, the threads in this example are not synchronized. Therefore, data can be lost or garbled if the producer places new data into the shared buffer before the consumer reads the previous data. Also, data can be incorrectly duplicated if the consumer consumes data again before the producer produces the next value. To show these possibilities, the consumer thread in the following example keeps a total of all the values it reads. If the consumer reads each value produced once and only once, the total will be 55. However, if you execute this program several times, you’ll see that the total is not always 55 (as shown in the outputs in Fig. 26.13). To emphasize the point, the producer and consumer threads in the example each sleep for random intervals of up to three seconds between performing their tasks. Thus, we do not know when the producer thread will attempt to write a new value, or when the consumer thread will attempt to read a value.
Implementing the Producer/Consumer Relationship

The program consists of interface Buffer (Fig. 26.9) and classes Producer (Fig. 26.10), Consumer (Fig. 26.11), UnsynchronizedBuffer (Fig. 26.12) and SharedBufferTest (Fig. 26.13). Interface Buffer (Fig. 26.9) declares methods set (line 6) and get (line 9) that a Buffer (such as UnsynchronizedBuffer) must implement to enable the Producer thread to place a value in the Buffer and the Consumer thread to retrieve a value from the Buffer, respectively. In subsequent examples, methods set and get will call methods that throw InterruptedExceptions. We declare each method with a throws clause here so that we don’t have to modify this interface for the later examples.

```java
public interface Buffer {
    // place int value into Buffer
    public void set(int value) throws InterruptedException;

    // return int value from Buffer
    public int get() throws InterruptedException;
}
```

Class Producer (Fig. 26.10) implements the Runnable interface, allowing it to be executed as a task in a separate thread. The constructor (lines 11–14) initializes the Buffer reference sharedLocation with an object created in main (line 14 of Fig. 26.13) and passed to the constructor. As we’ll see, this is an UnsynchronizedBuffer object that implements interface Buffer without synchronizing access to the shared object. The Producer thread in this program executes the tasks specified in the method run (lines 17–39). Each iteration of the loop (lines 21–35) invokes Thread method sleep (line 25) to place the Producer thread into the timed waiting state for a random time interval between 0 and 3 seconds. When the thread awakens, line 26 passes the value of control variable count to the Buffer object’s set method to set the shared buffer’s value. Lines 27–28 keep a total of all the values produced so far and output that value. When the loop completes, lines 37–38 display a message indicating that the Producer has finished producing data and is terminating. Next, method run terminates, which indicates that the Producer completed its task. Any method called from a Runnable’s run method (e.g., Buffer method set) executes as part of that task’s thread of execution. This fact becomes important in Sections 26.6–26.8 when we add synchronization to the producer/consumer relationship.

```java
public class Producer implements Runnable
{
    // Fig. 26.10: Producer.java
    // Producer with a run method that inserts the values 1 to 10 in buffer.
    import java.util.Random;

    public class Producer implements Runnable
    {
```

Fig. 26.9 | Buffer interface specifies methods called by Producer and Consumer.

Fig. 26.10 | Producer with a run method that inserts the values 1 to 10 in buffer. (Part 1 of 2.)
Class Consumer (Fig. 26.11) also implements interface Runnable, allowing the Consumer to execute concurrently with the Producer. Lines 11–14 initialize Buffer reference sharedLocation with an object that implements the Buffer interface (created in main, Fig. 26.13) and passed to the constructor as the parameter shared. As we’ll see, this is the same UnsynchronizedBuffer object that’s used to initialize the Producer object—thus, the two threads share the same object. The Consumer thread in this program performs the tasks specified in method run (lines 17–39). Lines 21–35 iterate 10 times. Each iteration invokes Thread method sleep (line 26) to put the Consumer thread into the timed waiting state for up to 3 seconds. Next, line 27 uses the Buffer’s get method to retrieve the value in the shared buffer, then adds the value to variable sum. Line 28 displays the total of all the values consumed so far. When the loop completes, lines 37–38 display a line indicating the sum of the consumed values. Then method run terminates, which indicates that the Consumer completed its task. Once both threads enter the terminated state, the program ends.
Note: We call method sleep in method run of the Producer and Consumer classes to emphasize the fact that, in multithreaded applications, it's unpredictable when each thread will perform its task and for how long it will perform the task when it has a processor. Normally, these thread scheduling issues are beyond the control of the Java developer. In this program, our thread's tasks are quite simple—the Producer writes the values 1 to 10 to the buffer, and the Consumer reads 10 values from the buffer and adds each value to variable sum. Without the sleep method call, and if the Producer executes first, given today's phenomenally fast processors, the Producer would likely complete its task before the Consumer got a chance to execute. If the Consumer executed first, it would likely consume garbage data ten times, then terminate before the Producer could produce the first real value.

```java
import java.util.Random;

public class Consumer implements Runnable {
    private final static Random generator = new Random();
    private final Buffer sharedLocation; // reference to shared object

    public Consumer( Buffer shared ) {
        sharedLocation = shared;
    }

    public void run() {
        int sum = 0;
        for ( int count = 1; count <= 10; count++ )
            try {
                Thread.sleep( generator.nextInt( 3000 ) );
                sum += sharedLocation.get();
                System.out.printf( "\t\t%2d
" , sum );
            } catch ( InterruptedException exception ) {
                exception.printStackTrace();
            }
        System.out.printf( "\nConsumer read values totaling %d\nTerminating Consumer\n" , sum );
    }
}
```

Fig. 26.11 Consumer with a run method that loops, reading 10 values from buffer.
26.5 Producer/Consumer Relationship without Synchronization

Class UnsynchronizedBuffer (Fig. 26.12) implements interface Buffer (line 4). An object of this class is shared between the Producer and the Consumer. Line 6 declares instance variable buffer and initializes it with the value –1. This value is used to demonstrate the case in which the Consumer attempts to consume a value before the Producer ever places a value in buffer. Methods set (lines 9–13) and get (lines 16–20) do not synchronize access to the field buffer. Method set simply assigns its argument to buffer (line 12), and method get simply returns the value of buffer (line 19).

```java
// Fig. 26.12: UnsynchronizedBuffer.java
// UnsynchronizedBuffer maintains the shared integer that is accessed by
// a producer thread and a consumer thread via methods set and get.
public class UnsynchronizedBuffer implements Buffer {
    private int buffer = -1; // shared by producer and consumer threads

    // place value into buffer
    public void set(int value) throws InterruptedException {
        System.out.printf("Producer writes\t%2d", value);
        buffer = value;
    } // end method set

    // return value from buffer
    public int get() throws InterruptedException {
        System.out.printf("Consumer reads\t%2d", buffer);
        return buffer;
    } // end method get
}
// end class UnsynchronizedBuffer
```

Fig. 26.12 | UnsynchronizedBuffer maintains the shared integer that is accessed by a producer thread and a consumer thread via methods set and get.

In class SharedBufferTest (Fig. 26.13), line 11 creates an ExecutorService to execute the Producer and Consumer Runnable s. Line 14 creates an UnsynchronizedBuffer object and assigns it to Buffer variable sharedLocation. This object stores the data that the Producer and Consumer threads will share. Lines 23–24 create and execute the Producer and Consumer. The Producer and Consumer constructors are each passed the same Buffer object (sharedLocation), so each object is initialized with a reference to the same Buffer. These lines also implicitly launch the threads and call each Runnable’s run method. Finally, line 26 calls method shutdown so that the application can terminate when the threads executing the Producer and Consumer complete their tasks. When main terminates (line 27), the main thread of execution enters the terminated state.

```java
// Fig. 26.13: SharedBufferTest.java
// Application with two threads manipulating an unsynchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```

Fig. 26.13 | Application with two threads manipulating an unsynchronized buffer. (Part 1 of 3.)
```java
public class SharedBufferTest {
    public static void main( String[] args ) {
        // create new thread pool with two threads
        ExecutorService application = Executors.newCachedThreadPool();
        // create UnsynchronizedBuffer to store ints
        Buffer sharedLocation = new UnsynchronizedBuffer();
        System.out.println( "Action	Value	Sum of Produced	Sum of Consumed"
        System.out.println( "------	-----	---------------	---------------
        // execute the Producer and Consumer, giving each of them access
        // to sharedLocation
        application.execute( new Producer( sharedLocation ) );
        application.execute( new Consumer( sharedLocation ) );
        application.shutdown(); // terminate application when tasks complete
    } // end main
} // end class SharedBufferTest
```

<table>
<thead>
<tr>
<th>Action</th>
<th>Value</th>
<th>Sum of Produced</th>
<th>Sum of Consumed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer writes 1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Producer writes 2</td>
<td>3</td>
<td>3</td>
<td>1 is lost</td>
</tr>
<tr>
<td>Producer writes 3</td>
<td>6</td>
<td>9</td>
<td>2 is lost</td>
</tr>
<tr>
<td>Consumer reads 3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Producer writes 4</td>
<td>10</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Consumer reads 4</td>
<td>10</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>Producer writes 5</td>
<td>15</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>Producer writes 6</td>
<td>21</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>Producer writes 7</td>
<td>28</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>Consumer reads 7</td>
<td>14</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>Consumer reads 7</td>
<td>21</td>
<td>112</td>
<td>7 read again</td>
</tr>
<tr>
<td>Producer writes 8</td>
<td>36</td>
<td>148</td>
<td></td>
</tr>
<tr>
<td>Consumer reads 8</td>
<td>30</td>
<td>178</td>
<td></td>
</tr>
<tr>
<td>Producer writes 9</td>
<td>45</td>
<td>223</td>
<td></td>
</tr>
<tr>
<td>Producer writes 10</td>
<td>55</td>
<td>278</td>
<td></td>
</tr>
</tbody>
</table>

Producer done producing
Terminating Producer
Consumer reads 10  47
Consumer reads 10  57  10 read again
Consumer reads 10  67  10 read again
Consumer reads 10  77  10 read again

Consumer read values totaling 77
Terminating Consumer

Fig. 26.13 | Application with two threads manipulating an unsynchronized buffer. (Part 2 of 3.)
26.5 Producer/Consumer Relationship without Synchronization

Recall from the overview of this example that we would like the Producer to execute first and every value produced by the Producer to be consumed exactly once by the Consumer. However, when we study the first output of Fig. 26.13, we see that the Producer writes the values 1, 2 and 3 before the Consumer reads its first value (3). Therefore, the values 1 and 2 are lost. Later, the values 5, 6 and 9 are lost, while 7 and 8 are read twice and 10 is read four times. So the first output produces an incorrect total of 77, instead of the correct total of 55. In the second output, the Consumer reads the value -1 before the Producer ever writes a value. The Consumer reads the value 1 five times before the Producer writes the value 2. Meanwhile, the values 5, 7, 8, 9 and 10 are all lost — the last four because the Consumer terminates before the Producer. An incorrect consumer total of 19 is displayed. (Lines in the output where the Producer or Consumer has acted out of order are highlighted.)

Error-Prevention Tip 26.1

Access to a shared object by concurrent threads must be controlled carefully or a program may produce incorrect results.

To solve the problems of lost and duplicated data, Section 26.6 presents an example in which we use an ArrayBlockingQueue (from package java.util.concurrent) to synchronize access to the shared object, guaranteeing that each and every value will be processed once and only once.
26.6 Producer/Consumer Relationship: ArrayBlockingQueue

One way to synchronize producer and consumer threads is to use classes from Java’s concurrency package that encapsulate the synchronization for you. Java includes the class ArrayBlockingQueue (from package java.util.concurrent)—a fully implemented, thread-safe buffer class that implements interface BlockingQueue. This interface extends the Queue interface discussed in Chapter 20 and declares methods put and take, the blocking equivalents of queue methods offer and poll, respectively. Method put places an element at the end of the BlockingQueue, waiting if the queue is full. Method take removes an element from the head of the BlockingQueue, waiting if the queue is empty. These methods make class ArrayBlockingQueue a good choice for implementing a shared buffer. Because method put blocks until there’s room in the buffer to write data, and method take blocks until there’s new data to read, the producer must produce a value first, the consumer correctly consumes only after the producer writes a value and the producer correctly produces the next value (after the first) only after the consumer reads the previous (or first) value. ArrayBlockingQueue stores the shared data in an array. The array’s size is specified as an argument to the ArrayBlockingQueue constructor. Once created, an ArrayBlockingQueue is fixed in size and will not expand to accommodate extra elements.

Figures 26.14–26.15 demonstrate a Producer and a Consumer accessing an ArrayBlockingQueue. Class BlockingBuffer (Fig. 26.14) uses an ArrayBlockingQueue object that stores an Integer (line 7). Line 11 creates the ArrayBlockingQueue and passes 1 to the constructor so that the object holds a single value, as we did with the UnsynchronizedBuffer of Fig. 26.12. Lines 7 and 11 use generics, which we discussed in Chapters 20–21. We discuss multiple-element buffers in Section 26.8. Because our BlockingBuffer class uses the thread-safe ArrayBlockingQueue class to manage access to the shared buffer, BlockingBuffer is itself thread safe, even though we have not implemented the synchronization ourselves.

```
1  // Fig. 26.14: BlockingBuffer.java
2  // Creating a synchronized buffer using an ArrayBlockingQueue.
3  import java.util.concurrent.ArrayBlockingQueue;
4  public class BlockingBuffer implements Buffer
5  {
6      private final ArrayBlockingQueue<Integer> buffer; // shared buffer
7
8      public BlockingBuffer()
9      {
10         buffer = new ArrayBlockingQueue<Integer>(1);
11     } // end BlockingBuffer constructor
12
13     // place value into buffer
14     public void set(int value) throws InterruptedException
15     {
16         buffer.put(value); // place value in buffer
17     }
```

Fig. 26.14 | Creating a synchronized buffer using an ArrayBlockingQueue. (Part 1 of 2.)
26.6 Producer/Consumer Relationship: ArrayBlockingQueue

ArrayBlockingQueue

BlockingBuffer implements interface Buffer (Fig. 26.9) and uses classes Producer (Fig. 26.10 modified to remove line 28) and Consumer (Fig. 26.11 modified to remove line 28) from the example in Section 26.5. This approach demonstrates that the threads accessing the shared object are unaware that their buffer accesses are now synchronized. The synchronization is handled entirely in the set and get methods of BlockingBuffer by calling the synchronized ArrayBlockingQueue methods put and take, respectively. Thus, the Producer and Consumer Runnables are properly synchronized simply by calling the shared object’s set and get methods.

Line 17 in method set (Fig. 26.14, lines 15–20) calls the ArrayBlockingQueue object’s put method. This method call blocks if necessary until there’s room in the buffer to place the value. Method get (lines 23–30) calls the ArrayBlockingQueue object’s take method (line 25). This method call blocks if necessary until there’s an element in the buffer to remove. Lines 18–19 and 26–27 use the ArrayBlockingQueue object’s size method to display the total number of elements currently in the ArrayBlockingQueue.

Class BlockingBufferTest (Fig. 26.15) contains the main method that launches the application. Line 12 creates an ExecutorService, and line 15 creates a BlockingBuffer object and assigns its reference to the Buffer variable sharedLocation. Lines 17–18 execute the Producer and Consumer Runnables. Line 19 calls method shutdown to end the application when the threads finish executing the Producer and Consumer tasks.
While methods put and take of ArrayBlockingQueue are properly synchronized, BlockingBuffer methods set and get (Fig. 26.14) are not declared to be synchronized. Thus, the statements performed in method set—the put operation (line 17) and the output (lines 18–19)—are not atomic; nor are the statements in method get—the take operation (line 25) and the output (lines 26–27). So there’s no guarantee that each output will occur immediately after the corresponding put or take operation, and the outputs may appear out of order. Even if they do, the ArrayBlockingQueue object is properly synchronizing access to the data, as evidenced by the fact that the sum of values read by the consumer is always correct.

```java
// create new thread pool with two threads
ExecutorService application = Executors.newCachedThreadPool();

// create BlockingBuffer to store ints
Buffer sharedLocation = new BlockingBuffer();

application.execute( new Producer( sharedLocation ) );
application.execute( new Consumer( sharedLocation ) );

application.shutdown();

} // end main

// end class BlockingBufferTest
```

## Fig. 26.15
Two threads manipulating a blocking buffer that properly implements the producer/consumer relationship. (Part 2 of 2.)
26.7 Producer/Consumer Relationship with Synchronization

The previous example showed how multiple threads can share a single-element buffer in a thread-safe manner by using the `ArrayBlockingQueue` class that encapsulates the synchronization necessary to protect the shared data. For educational purposes, we now explain how you can implement a shared buffer yourself using the `synchronized` keyword and methods of class `Object`. Using an `ArrayBlockingQueue` will result in more-maintainable and better-performing code.

The first step in synchronizing access to the buffer is to implement methods `get` and `set` as `synchronized` methods. This requires that a thread obtain the monitor lock on the `Buffer` object before attempting to access the buffer data, but it does not automatically ensure that threads proceed with an operation only if the buffer is in the proper state. We need a way to allow our threads to wait, depending on whether certain conditions are true. In the case of placing a new item in the buffer, the condition that allows the operation to proceed is that the buffer is not full. In the case of fetching an item from the buffer, the condition that allows the operation to proceed is that the buffer is not empty. If the condition in question is true, the operation may proceed; if it’s false, the thread must wait until it becomes true. When a thread is waiting on a condition, it’s removed from contention for the processor and placed into the waiting state and the lock it holds is released.

Methods `wait`, `notify` and `notifyAll`

Object methods `wait`, `notify` and `notifyAll`, which are inherited by all other classes, can be used with conditions to make threads `wait` when they cannot perform their tasks. If a thread obtains the monitor lock on an object, then determines that it cannot continue with its task on that object until some condition is satisfied, the thread can call `Object` method `wait` on the synchronized object; this releases the monitor lock on the object, and the thread waits in the waiting state while the other threads try to enter the object’s synchronized statement(s) or method(s). When a thread executing a synchronized statement (or method) completes or satisfies the condition on which another thread may be waiting, it can call `Object` method `notify` on the synchronized object to allow a waiting thread to transition to the runnable state again. At this point, the thread that was transitioned from the waiting state to the runnable state can attempt to reacquire the monitor lock on the object. Even if the thread is able to reacquire the monitor lock, it still might not be able to perform its task at this time—in which case the thread will reenter the waiting state and implicitly release the monitor lock. If a thread calls `notifyAll` on the synchronized object, then all the threads waiting for the monitor lock become eligible to reacquire the lock (that is, they all transition to the runnable state).

Remember that only one thread at a time can obtain the monitor lock on the object—other threads that attempt to acquire the same monitor lock will be blocked until the monitor lock becomes available again (i.e., until no other thread is executing in a synchronized statement on that object).

Common Programming Error 26.1

It’s an error if a thread issues a `wait`, a `notify` or a `notifyAll` on an object without having acquired a lock for it. This causes an `IllegalMonitorStateException`. 
Chapter 26  Multithreading

The application in Fig. 26.16 and Fig. 26.17 demonstrates a Producer and a Consumer accessing a shared buffer with synchronization. In this case, the Producer always produces a value first, the Consumer correctly consumes only after the Producer produces a value and the Producer correctly produces the next value only after the Consumer consumes the previous (or first) value. We reuse interface Buffer and classes Producer and Consumer from the example in Section 26.5, except that line 28 is removed from class Producer and class Consumer. The synchronization is handled in the set and get methods of class SynchronizedBuffer (Fig. 26.16), which implements interface Buffer (line 4). Thus, the Producer's and Consumer's run methods simply call the shared object's synchronized set and get methods.

```java
// Fig. 26.16: SynchronizedBuffer.java
// Synchronizing access to shared data using Object methods wait and notifyAll.
public class SynchronizedBuffer implements Buffer {
    private int buffer = -1; // shared by producer and consumer threads
    private boolean occupied = false; // whether the buffer is occupied

    // place value into buffer
    public synchronized void set(int value) throws InterruptedException {
        while (occupied)
            wait();

        buffer = value; // set new buffer value

        // indicate producer cannot store another value
        // until consumer retrieves current buffer value
        occupied = true;

        displayState("Producer writes " + buffer);

        notifyAll(); // tell waiting thread(s) to enter runnable state
    }

    // get value from buffer
    public synchronized int get() throws InterruptedException {
        while (occupied)
            wait();

        return buffer;
    }
}
```

**Error-Prevention Tip 26.2**

It's a good practice to use notifyAll to notify waiting threads to become runnable. Doing so avoids the possibility that your program would forget about waiting threads, which would otherwise starve.

The application in Fig. 26.16 and Fig. 26.17 demonstrates a Producer and a Consumer accessing a shared buffer with synchronization. In this case, the Producer always produces a value first, the Consumer correctly consumes only after the Producer produces a value and the Producer correctly produces the next value only after the Consumer consumes the previous (or first) value. We reuse interface Buffer and classes Producer and Consumer from the example in Section 26.5, except that line 28 is removed from class Producer and class Consumer. The synchronization is handled in the set and get methods of class SynchronizedBuffer (Fig. 26.16), which implements interface Buffer (line 4). Thus, the Producer's and Consumer's run methods simply call the shared object's synchronized set and get methods.
26.7 Producer/Consumer Relationship with Synchronization

Class SynchronizedBuffer contains fields buffer (line 6) and occupied (line 7). Methods set (lines 10–30) and get (lines 33–53) are declared as synchronized—only one thread can call either of these methods at a time on a particular SynchronizedBuffer object. Field occupied is used to determine whether it’s the Producer’s or the Consumer’s turn to perform a task. This field is used in conditional expressions in both the set and get methods. If occupied is false, then buffer is empty, so the Consumer cannot read the value of buffer, but the Producer can place a value into buffer. If occupied is true, the Consumer can read a value from buffer, but the Producer cannot place a value into buffer.

Method set and the Producer Thread

When the Producer thread’s run method invokes synchronized method set, the thread implicitly attempts to acquire the SynchronizedBuffer object’s monitor lock. If the monitor lock is available, the Producer thread implicitly acquires the lock. Then the loop at lines 13–19 first determines whether occupied is true. If so, buffer is full, so line 16 outputs a message indicating that the Producer thread is trying to write a value, and line 17 invokes method displayState (lines 56–60) to output another message indicating that buffer is empty. If occupied is false, line 19 outputs a message indicating that the Consumer thread is trying to read a value, and line 20 invokes method displayState to output another message indicating that buffer is empty.

Fields and Methods of Class SynchronizedBuffer

Class SynchronizedBuffer contains fields buffer (line 6) and occupied (line 7). Methods set (lines 10–30) and get (lines 33–53) are declared as synchronized—only one thread can call either of these methods at a time on a particular SynchronizedBuffer object. Field occupied is used to determine whether it’s the Producer’s or the Consumer’s turn to perform a task. This field is used in conditional expressions in both the set and get methods. If occupied is false, then buffer is empty, so the Consumer cannot read the value of buffer, but the Producer can place a value into buffer. If occupied is true, the Consumer can read a value from buffer, but the Producer cannot place a value into buffer.

Method get and the Consumer Thread

When the Consumer thread’s run method invokes synchronized method get, the thread implicitly attempts to acquire the SynchronizedBuffer object’s monitor lock. If the monitor lock is available, the Consumer thread implicitly acquires the lock. Then the loop at lines 41–59 first determines whether occupied is true. If so, buffer is empty, so line 49 outputs a message indicating that the Consumer thread is trying to read a value, and line 50 invokes method displayState to output another message indicating that buffer is empty. If occupied is false, line 52 outputs a message indicating that the Producer thread is trying to store another value.
full and that the Producer thread is waiting until there’s space. Line 18 invokes method wait (inherited from Object by SynchronizedBuffer) to place the thread that called method set (i.e., the Producer thread) in the waiting state for the SynchronizedBuffer object. The call to wait causes the calling thread to implicitly release the lock on the SynchronizedBuffer object. This is important because the thread cannot currently perform its task and because other threads (in this case, the Consumer) should be allowed to access the object to allow the condition (occupied) to change. Now another thread can attempt to acquire the SynchronizedBuffer object’s lock and invoke the object’s set or get method.

The Producer thread remains in the waiting state until another thread notifies the Producer that it may proceed—at which point the Producer returns to the runnable state and attempts to implicitly reacquire the lock on the SynchronizedBuffer object. If the lock is available, the Producer thread reacquires it, and method set continues executing with the next statement after the wait call. Because wait is called in a loop, the loop-continuation condition is tested again to determine whether the thread can proceed. If not, then wait is invoked again—otherwise, method set continues with the next statement after the loop.

Line 21 in method set assigns the value to the buffer. Line 25 sets occupied to true to indicate that the buffer now contains a value (i.e., a consumer can read the value, but a Producer cannot yet put another value there). Line 27 invokes method displayState to output a message indicating that the Producer is writing a new value into the buffer. Line 29 invokes method notifyAll (inherited from Object). If any threads are waiting on the SynchronizedBuffer object’s monitor lock, those threads enter the runnable state and can now attempt to reacquire the lock. Method notifyAll returns immediately, and method set then returns to the caller (i.e., the Producer’s run method). When method set returns, it implicitly releases the monitor lock on the SynchronizedBuffer object.

**Method get and the Consumer Thread**

Methods get and set are implemented similarly. When the Consumer thread’s run method invokes synchronized method get, the thread attempts to acquire the monitor lock on the SynchronizedBuffer object. If the lock is available, the Consumer thread acquires it. Then the while loop at lines 36–42 determines whether occupied is false. If so, the buffer is empty, so line 39 outputs a message indicating that the Consumer thread is trying to read a value, and line 40 invokes method displayState to output a message indicating that the buffer is empty and that the Consumer thread is waiting. Line 41 invokes method wait to place the thread that called method get (i.e., the Consumer) in the waiting state for the SynchronizedBuffer object. Again, the call to wait causes the calling thread to implicitly release the lock on the SynchronizedBuffer object, so another thread can attempt to acquire the SynchronizedBuffer object’s lock and invoke the object’s set or get method. If the lock on the SynchronizedBuffer is not available (e.g., if the Producer has not yet returned from method set), the Consumer is blocked until the lock becomes available.

The Consumer thread remains in the waiting state until it’s notified by another thread that it may proceed—at which point the Consumer thread returns to the runnable state and attempts to implicitly reacquire the lock on the SynchronizedBuffer object. If the lock is available, the Consumer reacquires it, and method get continues executing with the next statement after wait. Because wait is called in a loop, the loop-continuation condition is tested again to determine whether the thread can proceed with its execution. If not, wait is invoked again—otherwise, method get continues with the next statement after the loop.
Line 46 sets occupied to false to indicate that buffer is now empty (i.e., a Consumer cannot read the value, but a Producer can place another value in buffer), line 48 calls method displayState to indicate that the consumer is reading and line 50 invokes method notifyAll. If any threads are in the waiting state for the lock on this SynchronizedBuffer object, they enter the runnable state and can now attempt to reacquire the lock. Method notifyAll returns immediately, then method get returns the value of buffer to its caller. When method get returns, the lock on the SynchronizedBuffer object is implicitly released.

### Error-Prevention Tip 26.3
Always invoke method wait in a loop that tests the condition the task is waiting on. It's possible that a thread will reenter the runnable state (via a timed wait or another thread calling notifyAll) before the condition is satisfied. Testing the condition again ensures that the thread will not erroneously execute if it was notified early.

### Testing Class SynchronizedBuffer
Class SharedBufferTest2 (Fig. 26.17) is similar to class SharedBufferTest (Fig. 26.13). SharedBufferTest2 contains method main (lines 8–24), which launches the application. Line 11 creates an ExecutorService to run the Producer and Consumer tasks. Line 14 creates a SynchronizedBuffer object and assigns its reference to Buffer variable sharedLocation. This object stores the data that will be shared between the Producer and Consumer. Lines 16–17 display the column heads for the output. Lines 20–21 execute a Producer and a Consumer. Finally, line 23 calls method shutdown to end the application when the Producer and Consumer complete their tasks. When method main ends (line 24), the main thread of execution terminates.
```
23    application.shutdown();
24    } // end main
25    } // end class SharedBufferTest2
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Buffer</th>
<th>Occupied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer tries to read.</td>
<td>-1</td>
<td>false</td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Producer writes 1</td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 1</td>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>Consumer tries to read.</td>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Producer writes 2</td>
<td>2</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 2</td>
<td>2</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 3</td>
<td>3</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 3</td>
<td>3</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 4</td>
<td>4</td>
<td>true</td>
</tr>
<tr>
<td>Producer tries to write.</td>
<td>4</td>
<td>true</td>
</tr>
<tr>
<td>Buffer full. Producer waits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consumer reads 4</td>
<td>4</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 5</td>
<td>5</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 5</td>
<td>5</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 6</td>
<td>6</td>
<td>true</td>
</tr>
<tr>
<td>Producer tries to write.</td>
<td>6</td>
<td>true</td>
</tr>
<tr>
<td>Buffer full. Producer waits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consumer reads 6</td>
<td>6</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 7</td>
<td>7</td>
<td>true</td>
</tr>
<tr>
<td>Producer tries to write.</td>
<td>7</td>
<td>true</td>
</tr>
<tr>
<td>Buffer full. Producer waits.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consumer reads 7</td>
<td>7</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 8</td>
<td>8</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 8</td>
<td>8</td>
<td>false</td>
</tr>
<tr>
<td>Consumer tries to read.</td>
<td>8</td>
<td>false</td>
</tr>
<tr>
<td>Buffer empty. Consumer waits.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 26.17** | Two threads correctly manipulating a synchronized buffer. (Part 2 of 3.)
Study the outputs in Fig. 26.17. Observe that every integer produced is consumed exactly once—no values are lost, and no values are consumed more than once. The synchronization ensures that the Producer produces a value only when the buffer is empty and the Consumer consumes only when the buffer is full. The Producer always goes first, the Consumer waits if the Producer has not produced since the Consumer last consumed, and the Producer waits if the Consumer has not yet consumed the value that the Producer most recently produced. Execute this program several times to confirm that every integer produced is consumed exactly once. In the sample output, note the highlighted lines indicating when the Producer and Consumer must wait to perform their respective tasks.

26.8 Producer/Consumer Relationship: Bounded Buffers

The program in Section 26.7 uses thread synchronization to guarantee that two threads manipulate data in a shared buffer correctly. However, the application may not perform optimally. If the two threads operate at different speeds, one them will spend more (or most) of its time waiting. For example, in the program in Section 26.7 we shared a single integer variable between the two threads. If the Producer thread produces values faster than the Consumer can consume them, then the Producer thread waits for the Consumer, because there are no other locations in the buffer in which to place the next value. Similarly, if the Consumer consumes values faster than the Producer produces them, the Consumer waits until the Producer places the next value in the shared buffer. Even when we have threads that operate at the same relative speeds, those threads may occasionally become “out of sync” over a period of time, causing one of them to wait for the other. We cannot make assumptions about the relative speeds of concurrent threads—interactions that occur with the operating system, the network, the user and other components can cause the threads to operate at different and ever-changing speeds. When this happens, threads wait. When threads wait excessively, programs become less efficient, interactive programs become less responsive and applications suffer longer delays.
**Bounded Buffers**

To minimize the amount of waiting time for threads that share resources and operate at the same average speeds, we can implement a **bounded buffer** that provides a fixed number of buffer cells into which the **Producer** can place values, and from which the **Consumer** can retrieve those values. (In fact, we’ve already done this with the `ArrayBlockingQueue` class in Section 26.6.) If the **Producer** temporarily produces values faster than the **Consumer** can consume them, the **Producer** can write additional values into the extra buffer cells, if any are available. This capability enables the **Producer** to perform its task even though the **Consumer** is not ready to retrieve the current value being produced. Similarly, if the **Consumer** consumes faster than the **Producer** produces new values, the **Consumer** can read additional values (if there are any) from the buffer. This enables the **Consumer** to keep busy even though the **Producer** is not ready to produce additional values.

Even a **bounded buffer** is inappropriate if the **Producer** and the **Consumer** operate consistently at different speeds. If the **Consumer** always executes faster than the **Producer**, then a buffer containing one location is enough. Additional locations would simply waste memory. If the **Producer** always executes faster, only a buffer with an “infinite” number of locations would be able to absorb the extra production. However, if the **Producer** and **Consumer** execute at about the same average speed, a bounded buffer helps to smooth the effects of any occasional speeding up or slowing down in either thread’s execution.

The key to using a **bounded buffer** with a **Producer** and **Consumer** that operate at about the same speed is to provide the buffer with enough locations to handle the anticipated “extra” production. If, over a period of time, we determine that the **Producer** often produces as many as three more values than the **Consumer** can consume, we can provide a buffer of at least three cells to handle the extra production. Making the buffer too small would cause threads to wait longer; making the buffer too large would waste memory.

**Performance Tip 26.3**

Even when using a bounded buffer, it’s possible that a producer thread could fill the buffer, which would force the producer to wait until a consumer consumed a value to free an element in the buffer. Similarly, if the buffer is empty at any given time, a consumer thread must wait until the producer produces another value. The key to using a bounded buffer is to optimize the buffer size to minimize the amount of thread wait time, while not wasting space.

**Bounded Buffers Using ArrayBlockingQueue**

The simplest way to implement a bounded buffer is to use an `ArrayBlockingQueue` for the buffer so that all of the synchronization details are handled for you. This can be done by modifying the example from Section 26.6 to pass the desired size for the bounded buffer into the `ArrayBlockingQueue` constructor. Rather than repeat our previous `ArrayBlockingQueue` example with a different size, we instead present an example that illustrates how you can build a bounded buffer yourself. Again, using an `ArrayBlockingQueue` will result in more-maintainable and better-performing code. In Exercise 26.11, we ask you to reimplement this section’s example, using the Java Concurrency API techniques presented in Section 26.9.

**Implementing Your Own Bounded Buffer as a Circular Buffer**

The program in Fig. 26.18 and Fig. 26.19 demonstrates a **Producer** and a **Consumer** accessing a **bounded buffer with synchronization**. Again, we reuse interface `Buffer` and classes...
Producer and Consumer from the example in Section 26.5, except that line 28 is removed from class Producer and class Consumer. We implement the bounded buffer in class CircularBuffer (Fig. 26.18) as a circular buffer that uses a shared array of three elements. A circular buffer writes into and reads from the array elements in order, beginning at the first cell and moving toward the last. When a Producer or Consumer reaches the last element, it returns to the first and begins writing or reading, respectively, from there. In this version of the producer/consumer relationship, the Consumer consumes a value only when the array is not empty and the Producer produces a value only when the array is not full. The statements that created and started the thread objects in the main method of class SharedBufferTest2 (Fig. 26.17) now appear in class CircularBufferTest (Fig. 26.19).

```java
// Fig. 26.18: CircularBuffer.java
// Synchronizing access to a shared three-element bounded buffer.
public class CircularBuffer implements Buffer {

    private final int[] buffer = { -1, -1, -1 }; // shared buffer
    private int occupiedCells = 0; // count number of buffers used
    private int writeIndex = 0; // index of next element to write to
    private int readIndex = 0; // index of next element to read

    // place value into buffer
    public synchronized void set( int value ) throws InterruptedException {
        while ( occupiedCells == buffer.length ){
            System.out.printf( "Buffer is full. Producer waits.\n" );
            wait(); // wait until a buffer cell is free
        }
        buffer[ writeIndex ] = value; // set new buffer value
        ++occupiedCells; // one more buffer cell is full
        displayState( "Producer writes " + value );
        notifyAll(); // notify threads waiting to read from buffer
    }

    // return value from buffer
    public synchronized int get() throws InterruptedException {
        while ( occupiedCells == 0 ) {
            System.out.printf( "Buffer is empty. Consumer waits.\n" );
        }
        return buffer[ readIndex ]; // return buffer value
    }
}
```

Fig. 26.18 | Synchronizing access to a shared three-element bounded buffer. (Part 1 of 2.)
Chapter 26  Multithreading

wait(); // wait until a buffer cell is filled
} // end while

int readValue = buffer[ readIndex ]; // read value from buffer

// update circular read index
readIndex = ( readIndex + 1 ) % buffer.length;

--occupiedCells; // one fewer buffer cells are occupied
displayState( "Consumer reads " + readValue );
notifyAll(); // notify threads waiting to write to buffer

return readValue;
} // end method get

// display current operation and buffer state
public void displayState( String operation )
{

    // output operation and number of occupied buffer cells
    System.out.printf("%s\n(buffer cells occupied: ", occupiedCells, "buffer cells: ");

    for ( int value : buffer )
        System.out.printf(" %2d ", value ); // output values in buffer
    System.out.print("\n ");
    for ( int i = 0; i < buffer.length; i++ )
        System.out.print("---- ");
    System.out.println("\n ");

    for ( int i = 0; i < buffer.length; i++ )
    {
        if ( i == writeIndex && i == readIndex )
            System.out.print(" WR"); // both write and read index
        else if ( i == writeIndex )
            System.out.print(" W "); // just write index
        else if ( i == readIndex )
            System.out.print(" R "); // just read index
        else
            System.out.print(" "); // neither index
    } // end for

    System.out.println("\n");
} // end method displayState

public class CircularBuffer

Fig. 26.18  |  Synchronizing access to a shared three-element bounded buffer. (Part 2 of 2.)

Line 5 initializes array buffer as a three-element int array that represents the circular buffer. Variable occupiedCells (line 7) counts the number of elements in buffer that contain data to be read. When occupiedBuffers is 0, there’s no data in the circular buffer and the Consumer must wait—when occupiedCells is 3 (the size of the circular buffer),
the circular buffer is full and the Producer must wait. Variable writeIndex (line 8) indicates the next location in which a value can be placed by a Producer. Variable readIndex (line 9) indicates the position from which the next value can be read by a Consumer.

**CircularBuffer Method set**
CircularBuffer method set (lines 12–30) performs the same tasks as in Fig. 26.16, with a few modifications. The loop at lines 16–20 determines whether the Producer must wait (i.e., all buffer cells are full). If so, line 18 indicates that the Producer is waiting to perform its task. Then line 19 invokes method wait, causing the Producer thread to release the CircularBuffer's lock and wait until there's space for a new value to be written into the buffer. When execution continues at line 22 after the while loop, the value written by the Producer is placed in the circular buffer at location writeIndex. Then line 25 updates writeIndex for the next call to CircularBuffer method set. This line is the key to the buffer's circularity. When writeIndex is incremented past the end of the buffer, the line sets it to 0. Line 27 increments occupiedCells, because there's now one more value in the buffer that the Consumer can read. Next, line 28 invokes method displayState (lines 56–85) to update the output with the value produced, the number of occupied buffer cells, the contents of the buffer cells and the current writeIndex and readIndex. Line 29 invokes method notifyAll to transition waiting threads to the runnable state, so that a waiting Consumer thread (if there is one) can now try again to read a value from the buffer.

**CircularBuffer Method get**
CircularBuffer method get (lines 33–53) also performs the same tasks as it did in Fig. 26.16, with a few minor modifications. The loop at lines 37–41 determines whether the Consumer must wait (i.e., all buffer cells are empty). If the Consumer must wait, line 39 updates the output to indicate that the Consumer is waiting to perform its task. Then line 40 invokes method wait, causing the current thread to release the lock on the CircularBuffer and wait until data is available to read. When execution eventually continues at line 43 after a notifyAll call from the Producer, readValue is assigned the value at location readIndex in the circular buffer. Then line 46 updates readIndex for the next call to CircularBuffer method get. This line and line 25 implement the circularity of the buffer. Line 48 decrements occupiedCells, because there's now one more position in the buffer in which the Producer thread can place a value. Line 49 invokes method displayState to update the output with the consumed value, the number of occupied buffer cells, the contents of the buffer cells and the current writeIndex and readIndex. Line 50 invokes method notifyAll to allow any Producer threads waiting to write into the CircularBuffer object to attempt to write again. Then line 52 returns the consumed value to the caller.

**CircularBuffer Method displayState**
Method displayState (lines 56–85) outputs the application’s state. Lines 62–63 output the values of the buffer cells. Line 63 uses method printf with a "%2d" format specifier to print the contents of each buffer with a leading space if it's a single digit. Lines 70–82 output the current writeIndex and readIndex with the letters W and R, respectively.

**Testing Class CircularBuffer**
Class CircularBufferTest (Fig. 26.19) contains the main method that launches the application. Line 11 creates the ExecutorService, and line 14 creates a CircularBuffer ob-
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class CircularBufferTest {
    public static void main( String[] args ){
        // create new thread pool with two threads
        ExecutorService application = Executors.newCachedThreadPool();

        // create CircularBuffer to store ints
        CircularBuffer sharedLocation = new CircularBuffer();

        // display the initial state of the CircularBuffer
        sharedLocation.displayState( "Initial State" );

        // execute the Producer and Consumer tasks
        application.execute( new Producer( sharedLocation ) );
        application.execute( new Consumer( sharedLocation ) );

        application.shutdown();
    }
}

Initial State (buffer cells occupied: 0)
buffer cells:  -1  -1  -1
             ---- ---- ----
            WR

Producer writes 1 (buffer cells occupied: 1)
buffer cells:   1  -1  -1
              ---- ---- ----
             R  W
<table>
<thead>
<tr>
<th>Operation</th>
<th>Buffer State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumer reads 1</td>
<td>buffer: 1 -1 -1</td>
</tr>
<tr>
<td>Producer writes 2</td>
<td>buffer: 1 2 -1</td>
</tr>
<tr>
<td>Consumer reads 2</td>
<td>buffer: 1 2 -1</td>
</tr>
<tr>
<td>Producer writes 3</td>
<td>buffer: 1 2 3</td>
</tr>
<tr>
<td>Consumer reads 3</td>
<td>buffer: 1 2 3</td>
</tr>
<tr>
<td>Producer writes 4</td>
<td>buffer: 4 2 3</td>
</tr>
<tr>
<td>Producer writes 5</td>
<td>buffer: 4 5 3</td>
</tr>
<tr>
<td>Consumer reads 4</td>
<td>buffer: 4 5 3</td>
</tr>
<tr>
<td>Producer writes 6</td>
<td>buffer: 4 5 6</td>
</tr>
<tr>
<td>Producer writes 7</td>
<td>buffer: 7 5 6</td>
</tr>
<tr>
<td>Consumer reads 5</td>
<td>buffer: 7 5 6</td>
</tr>
<tr>
<td>Producer writes 8</td>
<td>buffer: 7 8 6</td>
</tr>
</tbody>
</table>

Fig. 26.19 | Producer and Consumer threads manipulating a circular buffer. (Part 2 of 3.)
26.9 Producer/Consumer Relationship: The Lock and Condition Interfaces

Though the synchronized keyword provides for most basic thread-synchronization needs, Java provides other tools to assist in developing concurrent programs. In this section, we discuss the Lock and Condition interfaces. These interfaces give you more precise control over thread synchronization, but are more complicated to use.

**Interface Lock and Class ReentrantLock**

Any object can contain a reference to an object that implements the Lock interface (of package java.util.concurrent.locks). A thread calls the Lock's `lock` method (analogous to entering a synchronized block) to acquire the lock. Once a Lock has been obtained by one thread, the Lock object will not allow another thread to obtain the Lock until the first thread releases the Lock (by calling the Lock's `unlock` method—analogous to ex-
26.9 The Lock and Condition Interfaces

It is a synchronized block. If several threads are trying to call method lock on the same Lock object at the same time, only one of these threads can obtain the lock—all the others are placed in the waiting state for that lock. When a thread calls method unlock, the lock on the object is released and a waiting thread attempting to lock the object proceeds.

Class ReentrantLock (of package java.util.concurrent.locks) is a basic implementation of the Lock interface. The constructor for a ReentrantLock takes a boolean argument that specifies whether the lock has a fairness policy. If the argument is true, the ReentrantLock’s fairness policy is “the longest-waiting thread will acquire the lock when it’s available.” Such a fairness policy guarantees that indefinite postponement (also called starvation) cannot occur. If the fairness policy argument is set to false, there’s no guarantee as to which waiting thread will acquire the lock when it’s available.

Software Engineering Observation 26.3
Using a ReentrantLock with a fairness policy avoids indefinite postponement.

Performance Tip 26.4
Using a ReentrantLock with a fairness policy can decrease program performance.

Condition Objects and Interface Condition
If a thread that owns a Lock determines that it cannot continue with its task until some condition is satisfied, the thread can wait on a condition object. Using Lock objects allows you to explicitly declare the condition objects on which a thread may need to wait. For example, in the producer/consumer relationship, producers can wait on one object and consumers can wait on another. This is not possible when using the synchronized keywords and an object’s built-in monitor lock. Condition objects are associated with a specific Lock and are created by calling a Lock’s newCondition method, which returns an object that implements the Condition interface (of package java.util.concurrent.locks). To wait on a condition object, the thread can call the Condition’s await method (analogous to Object method wait). This immediately releases the associated Lock and places the thread in the waiting state for that Condition. Other threads can then try to obtain the Lock. When a runnable thread completes a task and determines that the waiting thread can now continue, the runnable thread can call Condition method signal (analogous to Object method notify) to allow a thread in that Condition’s waiting state to return to the runnable state. At this point, the thread that transitioned from the waiting state to the runnable state can attempt to reacquire the Lock. Even if it’s able to reacquire the Lock, the thread still might not be able to perform its task at this time—in which case the thread can call the Condition’s await method to release the Lock and reenter the waiting state. If multiple threads are in a Condition’s waiting state when signal is called, the default implementation of Condition signals the longest-waiting thread to transition to the runnable state. If a thread calls Condition method signalAll (analogous to Object method notifyAll), then all the threads waiting for that condition transition to the runnable state and become eligible to reacquire the Lock. Only one of those threads can obtain the Lock on the object—the others will wait until the Lock becomes available again. If the Lock has a fairness policy, the longest-waiting thread acquires the Lock. When a thread is finished with a shared object, it must call method unlock to release the Lock.
Chapter 26  Multithreading

Lock and Condition vs. the synchronized Keyword

In some applications, using Lock and Condition objects may be preferable to using the synchronized keyword. Locks allow you to interrupt waiting threads or to specify a timeout for waiting to acquire a lock, which is not possible using the synchronized keyword. Also, a Lock is not constrained to be acquired and released in the same block of code, which is the case with the synchronized keyword. Condition objects allow you to specify multiple conditions on which threads may wait. Thus, it’s possible to indicate to waiting threads that a specific condition object is now true by calling signal or signalAll on that Condition object. With synchronized, there’s no way to explicitly state the condition on which threads are waiting, and thus there’s no way to notify threads waiting on one condition that they may proceed without also signaling threads waiting on any other conditions. There are other possible advantages to using Lock and Condition objects, but generally it’s best to use the synchronized keyword unless your application requires advanced synchronization capabilities.

Using Locks and Conditions to Implement Synchronization

To illustrate how to use the Lock and Condition interfaces, we now implement the producer/consumer relationship using Lock and Condition objects to coordinate access to a shared single-element buffer (Fig. 26.20 and Fig. 26.21). In this case, each produced value is correctly consumed exactly once. Again, we reuse interface Buffer and classes Producer and Consumer from the example in Section 26.5, except that line 28 is removed from class Producer and class Consumer.
Class SynchronizedBuffer (Fig. 26.20) contains five fields. Line 11 creates a new object of type ReentrantLock and assigns its reference to Lock variable accessLock. The ReentrantLock is created without the *fairness policy* because at any time only a single Producer or Consumer will be waiting to acquire the Lock in this example. Lines 14–15 create two Conditions using Lock method newCondition. Condition canWrite contains a queue for a Producer thread waiting while the buffer is full (i.e., there’s data in the buffer that the Consumer has not read yet). If the buffer is full, the Producer calls method await on this Condition. When the Consumer reads data from a full buffer, it calls method signal on this Condition. Condition canRead contains a queue for a Consumer thread waiting while the buffer is empty (i.e., there’s no data in the buffer for the Consumer to read). If the buffer is empty, the Consumer calls method await on this Condition. When the Producer writes to the empty buffer, it calls method signal on this Condition. The int variable buffer (line 17) holds the shared data. The boolean variable occupied (line 18) keeps track of whether the buffer currently holds data (that the Consumer should read).

```java
// Fig. 26.20: SynchronizedBuffer.java
// Synchronizing access to a shared integer using the Lock and Condition interfaces
// import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

public class SynchronizedBuffer implements Buffer {

    private boolean occupied = false;

    // place int value into buffer
    public void set(int value) throws InterruptedException {
        try {
            // while buffer is not empty, place thread in waiting state
            while (occupied) {
                System.out.println("Producer tries to write. ");
                displayState("Buffer full. Producer waits.");
                canWrite.await(); // wait until buffer is empty
            }
        }
        finally {
            accessLock.lock(); // lock this object
        }
    }

    // output thread information and buffer information, then wait
    try {
        // while buffer is not empty, place thread in waiting state
        while (occupied) {
            System.out.println("Producer tries to write. ");
            displayState("Buffer full. Producer waits.");
            canWrite.await(); // wait until buffer is empty
        }
    }
```

*Fig. 26.20* | Synchronizing access to a shared integer using the Lock and Condition interfaces. (Part 1 of 3.)
buffer = value; // set new buffer value

// indicate producer cannot store another value
// until consumer retrieves current buffer value
occupied = true;

displayState( "Producer writes " + buffer);

  // signal any threads waiting to read from buffer
  canRead.signalAll();
}

}[// end try
finally
{
  accessLock.unlock(); // unlock this object
}
// end finally
}
// end method set


public int get() throws InterruptedException
{
  int readValue = 0; // initialize value read from buffer

  accessLock.lock(); // lock this object

  // output thread information and buffer information, then wait
  try
  {
    // if there is no data to read, place thread in waiting state
    while ( !occupied )
    {
      System.out.println( "Consumer tries to read." );
      displayState( "Buffer empty. Consumer waits." );
      canRead.await(); // wait until buffer is full
    }
    // end while

    // indicate that producer can store another value
    // because consumer just retrieved buffer value
    occupied = false;

    readValue = buffer; // retrieve value from buffer
    displayState( "Consumer reads " + readValue );

    // signal any threads waiting for buffer to be empty
    canWrite.signalAll();
  }
  // end try

  finally
  {
    accessLock.unlock(); // unlock this object
  }
  // end finally

  return readValue;
}
// end method get
26.9 The Lock and Condition Interfaces

Line 23 in method set calls method lock on the SynchronizedBuffer’s accessLock. If the lock is available (i.e., no other thread has acquired it), this thread now owns the lock and the thread continues. If the lock is unavailable (i.e., it’s held by another thread), method lock waits until the lock is released. After the lock is acquired, lines 26–46 execute. Line 29 tests occupied to determine whether buffer is full. If it is, lines 31–32 display a message indicating that the thread will wait. Line 33 calls Condition method await on the canWrite condition object, which temporarily releases the SynchronizedBuffer’s Lock and waits for a signal from the Consumer that buffer is available for writing. When buffer is available, the method proceeds, writing to buffer (line 36), setting occupied to true (line 40) and displaying a message indicating that the producer wrote a value (line 42). Line 45 calls Condition method signal on condition object canRead to notify the waiting Consumer (if there is one) that the buffer has new data to be read. Line 49 calls method unlock from a finally block to release the lock and allow the Consumer to proceed.

Line 57 of method get (lines 54–86) calls method lock to acquire the Lock. This method waits until the Lock is available. Once the Lock is acquired, line 63 tests whether occupied is false, indicating that the buffer is empty. If so, line 67 calls method await on condition object canRead. Recall that method signal is called on variable canRead in the set method (line 45). When the Condition object is signaled, the get method continues. Line 72–74 set occupied to false, store the value of buffer in readValue and output the readValue. Then line 78 signals the condition object canWrite. This awakens the Producer if it’s indeed waiting for the buffer to be emptied. Line 82 calls method unlock from a finally block to release the lock, and line 85 returns readValue to the caller.

Class SharedBufferTest2 (Fig. 26.21) is identical to that of Fig. 26.17. Study the outputs in Fig. 26.21. Observe that every integer produced is consumed exactly once—no values are lost, and no values are consumed more than once. The Lock and Condition objects ensure that the Producer and Consumer cannot perform their tasks unless it’s their turn.
Chapter 26  Multithreading

The Producer must go first, the Consumer must wait if the Producer has not produced since the Consumer last consumed and the Producer must wait if the Consumer has not yet consumed the value that the Producer most recently produced. Execute this program several times to confirm that every integer produced is consumed exactly once. In the sample output, note the highlighted lines indicating when the Producer and Consumer must wait to perform their respective tasks.

```
// Fig. 26.21: SharedBufferTest2.java
// Two threads manipulating a synchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SharedBufferTest2 {
    public static void main( String[] args ) {
        // create new thread pool with two threads
        ExecutorService application = Executors.newCachedThreadPool();

        // create SynchronizedBuffer to store ints
        Buffer sharedLocation = new SynchronizedBuffer();

        // execute the Producer and Consumer tasks
        application.execute( new Producer( sharedLocation ) );
        application.execute( new Consumer( sharedLocation ) );

        application.shutdown();
    }
}
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>Buffer</th>
<th>Occupied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer writes 1</td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>Producer tries to write.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer full. Producer waits.</td>
<td>1</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 1</td>
<td>1</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 2</td>
<td>2</td>
<td>true</td>
</tr>
<tr>
<td>Producer tries to write.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buffer full. Producer waits.</td>
<td>2</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 2</td>
<td>2</td>
<td>false</td>
</tr>
<tr>
<td>Producer writes 3</td>
<td>3</td>
<td>true</td>
</tr>
<tr>
<td>Consumer reads 3</td>
<td>3</td>
<td>false</td>
</tr>
</tbody>
</table>

**Fig. 26.21** | Two threads manipulating a synchronized buffer. (Part 1 of 2.)
26.10 Concurrent Collections Overview

In Chapter 20, we introduced various collections from the Java Collections API. We also mentioned that you can obtain synchronized versions of those collections to allow only one thread at a time to access a collection that might be shared among several threads. The collections from the java.util.concurrent package are specifically designed and optimized for use in programs that share collections among multiple threads.

Figure 26.22 lists the many concurrent collections in package java.util.concurrent. For more information on these collections, visit

```
download.oracle.com/javase/6/docs/api/java/util/concurrent/
package-summary.html
```
For information on the additional concurrent collections that are new in Java SE 7, visit 
download.java.net/jdk7/docs/api/java/util/concurrent/
package-summary.html

<table>
<thead>
<tr>
<th>Collection</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayBlockingQueue</td>
<td>A fixed-size queue that supports the producer/consumer relationship—possibly with many producers and consumers.</td>
</tr>
<tr>
<td>ConcurrentHashMap</td>
<td>A hash-based map that allows an arbitrary number of reader threads and a limited number of writer threads.</td>
</tr>
<tr>
<td>ConcurrentLinkedQueue</td>
<td>A concurrent linked-list implementation of a queue that can grow dynamically.</td>
</tr>
<tr>
<td>ConcurrentSkipListMap</td>
<td>A concurrent map that is sorted by its keys.</td>
</tr>
<tr>
<td>ConcurrentSkipListSet</td>
<td>A sorted concurrent set.</td>
</tr>
<tr>
<td>CopyOnWriteArrayList</td>
<td>A thread-safe ArrayList. Each operation that modifies the collection first creates a new copy of the contents. Used when the collection is traversed much more frequently than the collection's contents are modified.</td>
</tr>
<tr>
<td>CopyOnWriteArraySet</td>
<td>A set that's implemented using CopyOnWriteArrayList.</td>
</tr>
<tr>
<td>DelayQueue</td>
<td>A variable-size queue containing Delayed objects. An object can be removed only after its delay has expired.</td>
</tr>
<tr>
<td>LinkedBlockingDeque</td>
<td>A double-ended blocking queue implemented as a linked list that can optionally be fixed in size.</td>
</tr>
<tr>
<td>LinkedBlockingQueue</td>
<td>A blocking queue implemented as a linked list that can optionally be fixed in size.</td>
</tr>
<tr>
<td>PriorityBlockingQueue</td>
<td>A variable-length priority-based blocking queue (like a PriorityQueue).</td>
</tr>
<tr>
<td>SynchronousQueue</td>
<td>A blocking queue implementation that does not have an internal capacity. Each insert operation by one thread must wait for a remove operation from another thread and vice versa.</td>
</tr>
</tbody>
</table>

**Concurrent Collections Added in Java SE 7**

<table>
<thead>
<tr>
<th>Collection</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConcurrentLinkedDeque</td>
<td>A concurrent linked-list implementation of a double-ended queue.</td>
</tr>
<tr>
<td>LinkedTransferQueue</td>
<td>A linked-list implementation of interface TransferQueue. Each producer has the option of waiting for a consumer to take an element being inserted (via method transfer) or simply placing the element into the queue (via method put). Also provides overloaded method tryTransfer to immediately transfer an element to a waiting consumer or to do so within a specified timeout period. If the transfer cannot be completed, the element is not placed in the queue. Typically used in applications that pass messages between threads.</td>
</tr>
</tbody>
</table>

Fig. 26.22 | Concurrent collections summary (package java.util.concurrent).
Swing applications present a unique set of challenges for multithreaded programming. All Swing applications have a single thread, called the event dispatch thread, to handle interactions with the application’s GUI components. Typical interactions include updating GUI components or processing user actions such as mouse clicks. All tasks that require interaction with an application’s GUI are placed in an event queue and are executed sequentially by the event dispatch thread.

Swing GUI components are not thread safe—they cannot be manipulated by multiple threads without the risk of incorrect results. Unlike the other examples presented in this chapter, thread safety in GUI applications is achieved not by synchronizing thread actions, but by ensuring that Swing components are accessed from only a single thread—the event dispatch thread. This technique is called thread confinement. Allowing just one thread to access non-thread-safe objects eliminates the possibility of corruption due to multiple threads accessing these objects concurrently.

Usually it’s sufficient to perform simple calculations on the event dispatch thread in sequence with GUI component manipulations. If an application must perform a lengthy computation in response to a user interface interaction, the event dispatch thread cannot attend to other tasks in the event queue while the thread is tied up in that computation. This causes the GUI components to become unresponsive. It’s preferable to handle a long-running computation in a separate thread, freeing the event dispatch thread to continue managing other GUI interactions. Of course, to update the GUI based on the computation’s results, you must update the GUI from the event dispatch thread, rather than from the worker thread that performed the computation.

Class SwingWorker

Class SwingWorker (in package javax.swing) perform long-running computations in a worker thread and to update Swing components from the event dispatch thread based on the computations’ results. SwingWorker implements the Runnable interface, meaning that a SwingWorker object can be scheduled to execute in a separate thread. The SwingWorker class provides several methods to simplify performing computations in a worker thread and making the results available for display in a GUI. Some common SwingWorker methods are described in Fig. 26.23.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>doInBackground</td>
<td>Defines a long computation and is called in a worker thread.</td>
</tr>
<tr>
<td>done</td>
<td>Executes on the event dispatch thread when doInBackground returns.</td>
</tr>
<tr>
<td>execute</td>
<td>Schedules the SwingWorker object to be executed in a worker thread.</td>
</tr>
<tr>
<td>get</td>
<td>Waits for the computation to complete, then returns the result of the computation (i.e., the return value of doInBackground).</td>
</tr>
<tr>
<td>publish</td>
<td>Sends intermediate results from the doInBackground method to the process method for processing on the event dispatch thread.</td>
</tr>
</tbody>
</table>

Fig. 26.23 | Commonly used SwingWorker methods. (Part 1 of 2.)
Chapter 26  Multithreading

26.1.1 Performing Computations in a Worker Thread

In the next example, the user enters a number \( n \) and the program gets the \( n \)th Fibonacci number, which we calculate using the recursive algorithm discussed in Section 18.4. Since the algorithm is time consuming for large values, we use a SwingWorker object to perform the calculation in a worker thread. The GUI also provides a separate set of components that get the next Fibonacci number in the sequence with each click of a button, beginning with \( \text{fibonacci}(1) \). This set of components performs its short computation directly in the event dispatch thread. This program is capable of producing up to the 92nd Fibonacci number—subsequent values are outside the range that can be represented by a long. Recall that you can use class BigInteger to represent arbitrarily large integer values.

Class BackgroundCalculator (Fig. 26.24) performs the recursive Fibonacci calculation in a \textit{worker thread}. This class extends SwingWorker (line 8), overriding the methods doInBackground and done. Method doInBackground (lines 21–24) computes the \( n \)th Fibonacci number in a worker thread and returns the result. Method done (lines 27–43) displays the result in a JLabel.

```
1 // Fig. 26.24: BackgroundCalculator.java
2 // SwingWorker subclass for calculating Fibonacci numbers
3 // in a background thread.
4 import javax.swing.SwingWorker;
5 import javax.swing.JLabel;
6 import java.util.concurrent.ExecutionException;
7 public class BackgroundCalculator extends SwingWorker<Long, Object>
8 {
9     private final int n; // Fibonacci number to calculate
10     private final JLabel resultJLabel; // JLabel to display the result
11
12     // constructor
13     public BackgroundCalculator( int number, JLabel label )
14     {
15         n = number;
16         resultJLabel = label;
17     } // end BackgroundCalculator constructor
```

Fig. 26.23  |  Commonly used SwingWorker methods. (Part 2 of 2.)
Multithreading with GUI

SwingWorker is a generic class. In line 8, the first type parameter is Long and the second is Object. The first type parameter indicates the type returned by the doInBackground method; the second indicates the type that's passed between the publish and process methods to handle intermediate results. Since we do not use publish and process in this example, we simply use Object as the second type parameter. We discuss publish and process in Section 26.11.2.

A BackgroundCalculator object can be instantioted from a class that controls a GUI. A BackgroundCalculator maintains instance variables for an integer that represents the Fibonacci number to be calculated and a JLabel that displays the results of the calculation (lines 10–11). The BackgroundCalculator constructor (lines 14–18) initializes these instance variables with the arguments that are passed to the constructor.
1098  Chapter 26  Multithreading

Software Engineering Observation 26.4

Any GUI components that will be manipulated by SwingWorker methods, such as components that will be updated from methods process or done, should be passed to the SwingWorker subclass’s constructor and stored in the subclass object. This gives these methods access to the GUI components they’ll manipulate.

When method execute is called on a BackgroundCalculator object, the object is scheduled for execution in a worker thread. Method doInBackground is called from the worker thread and invokes the fibonacci method (lines 46–52), passing instance variable n as an argument (line 23). Method fibonacci uses recursion to compute the Fibonacci of n. When fibonacci returns, method doInBackground returns the result.

After doInBackground returns, method done is called from the event dispatch thread. This method attempts to set the result JLabel to the return value of doInBackground by calling method get to retrieve this return value (line 32). Method get waits for the result to be ready if necessary, but since we call it from method done, the computation will be complete before get is called. Lines 34–37 catch InterruptedException if the current thread is interrupted while waiting for get to return. This exception will not occur in this example since the calculation will have already completed by the time get is called. Lines 38–42 catch ExecutionException, which is thrown if an exception occurs during the computation.

Class FibonacciNumbers

Class FibonacciNumbers (Fig. 26.25) displays a window containing two sets of GUI components—one set to compute a Fibonacci number in a worker thread and another to get the next Fibonacci number in response to the user’s clicking a JButton. The constructor (lines 38–109) places these components in separate titled JPanels. Lines 46–47 and 78–79 add two JLabels, a JTextField and a JButton to the worker JPanel to allow the user to enter an integer whose Fibonacci number will be calculated by the BackgroundWorker. Lines 84–85 and 103 add two JLabels and a JButton to the event dispatch thread panel to allow the user to get the next Fibonacci number in the sequence. Instance variables n1 and n2 contain the previous two Fibonacci numbers in the sequence and are initialized to 0 and 1, respectively (lines 29–30). Instance variable count stores the most recently computed sequence number and is initialized to 1 (line 31). The two JLabels display count and n2 initially, so that the user will see the text Fibonacci of 1: 1 in the eventThreadPanel when the GUI starts.

Fig. 26.25  Using SwingWorker to perform a long calculation with results displayed in a GUI.

(Part 1 of 4.)
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.border.TitledBorder;
import javax.swing.border.LineBorder;
import java.awt.Color;
import java.util.concurrent.ExecutionException;

public class FibonacciNumbers extends JFrame {
    // components for calculating the Fibonacci of a user-entered number
    private final JPanel workerJPanel = new JPanel(new GridLayout(2, 2, 5, 5));
    private final JTextField numberJTextField = new JTextField();
    private final JButton goJButton = new JButton("Go");
    private final JLabel fibonacciJLabel = new JLabel();

    // components and variables for getting the next Fibonacci number
    private final JPanel eventThreadJPanel = new JPanel(new GridLayout(2, 2, 5, 5));
    private long n1 = 0; // initialize with first Fibonacci number
    private long n2 = 1; // initialize with second Fibonacci number
    private int count = 1; // current Fibonacci number to display
    private final JLabel nJLabel = new JLabel("Fibonacci of 1: ");
    private final JLabel nFibonacciJLabel = new JLabel(String.valueOf(n2));
    private final JButton nextNumberJButton = new JButton("Next Number");

    // constructor
    public FibonacciNumbers() {
        super("Fibonacci Numbers");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new GridLayout(2, 1, 10, 10));

        // add GUI components to the SwingWorker panel
        workerJPanel.setBorder(new TitledBorder(new LineBorder(Color.BLACK), "With SwingWorker"));
        workerJPanel.add(new JLabel("Get Fibonacci of:"));
        workerJPanel.add(numberJTextField);
        goJButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                int n;
                try {
                    // retrieve user’s input as an integer
                    n = Integer.parseInt(numberJTextField.getText());
                } catch (NumberFormatException ex) {
                    // end try
                }
                catch( NumberFormatException ex )
            }
        });
    }

    Fig. 26.25 | Using SwingWorker to perform a long calculation with results displayed in a GUI.
    (Part 2 of 4.)
// display an error message if the user did not enter an integer
fibonacciJLabel.setText( "Enter an integer." );
return;
} // end catch

// indicate that the calculation has begun
fibonacciJLabel.setText( "Calculating..." );

// create a task to perform calculation in background
BackgroundCalculator task =
    new BackgroundCalculator( n, fibonacciJLabel );
task.execute(); // execute the task

} // end method actionPerformed
} // end anonymous inner class
}

// add GUI components to the event-dispatching thread panel
eventThreadJPanel.setBorder( new TitledBorder( new LineBorder( Color.BLACK ), "Without SwingWorker" ) );
eventThreadJPanel.add( nJLabel );
eventThreadJPanel.add( nFibonacciJLabel );
nextNumberJButton.addActionListener( new ActionListener()
{
    public void actionPerformed( ActionEvent event )
    {
        // calculate the Fibonacci number after n2
        long temp = n1 + n2;
        n1 = n2;
        n2 = temp;
        ++count;

        // display the next Fibonacci number
        nJLabel.setText( "Fibonacci of " + count + ": " );
        nFibonacciJLabel.setText( String.valueOf( n2 ) );
    } // end method actionPerformed
} // end anonymous inner class
}; // end call to addActionListener
eventThreadJPanel.add( nextNumberJButton );

add( workerJPanel );
add( eventThreadJPanel );
setSize( 275, 200 );
setVisible( true );
} // end constructor

// main method begins program execution
public static void main( String[] args )
{
26.11 Multithreading with GUI

Lines 48–77 register the event handler for the goJButton. If the user clicks this JButton, line 58 gets the value entered in the numberJTextField and attempts to parse it as an integer. Lines 72–73 create a new BackgroundCalculator object, passing in the user-entered value and the fibonacciJLabel that’s used to display the calculation’s results. Line 74 calls method execute on the BackgroundCalculator, scheduling it for execution in a separate worker thread. Method execute does not wait for the BackgroundCalculator to finish executing. It returns immediately, allowing the GUI to continue processing other events while the computation is performed.

If the user clicks the nextNumberJButton in the eventThreadJPanel, the event handler registered in lines 86–102 executes. Lines 92–95 add the previous two Fibonacci numbers stored in n1 and n2 to determine the next number in the sequence, update n1 and n2 to their new values and increment count. Then lines 98–99 update the GUI to display the next number. The code for these calculations is in method actionPerformed, so they’re performed on the event dispatch thread. Handling such short computations in the event dispatch thread does not cause the GUI to become unresponsive, as with the recursive algorithm for calculating the Fibonacci of a large number. Because the longer Fibonacci computation is performed in a separate worker thread using the SwingWorker, it’s possible to get the next Fibonacci number while the recursive computation is still in progress.
26.11.2 Processing Intermediate Results with SwingWorker

We’ve presented an example that uses the SwingWorker class to execute a long process in a background thread and update the GUI when the process is finished. We now present an example of updating the GUI with intermediate results before the long process completes. Figure 26.26 presents class PrimeCalculator, which extends SwingWorker to compute the first \( n \) prime numbers in a worker thread. In addition to the doInBackground and done methods used in the previous example, this class uses SwingWorker methods publish, process and setProgress. In this example, method publish sends prime numbers to method process as they’re found, method process displays these primes in a GUI component and method setProgress updates the progress property. We later show how to use this property to update a JProgressBar.

```java
// Fig. 26.26: PrimeCalculator.java
// Calculates the first n primes, displaying them as they are found.
import javax.swing.JTextArea;
import javax.swing.JLabel;
import javax.swing.JButton;
import javax.swing.SwingWorker;
import java.util.Arrays;
import java.util.Random;
import java.util.List;
import java.util.concurrentCancellationException;
import java.util.concurrent.ExecutionException;

public class PrimeCalculator extends SwingWorker<Integer, Integer> {
    private final Random generator = new Random();
    private final JTextArea intermediateJTextArea;
    private final JButton getPrimesJButton;
    private final JButton cancelJButton;
    private final JLabel statusJLabel;
    private final boolean[] primes;

    public PrimeCalculator(int max, JTextArea intermediate, JLabel status,
                            JButton getPrimes, JButton cancel) {
        intermediateJTextArea = intermediate;
        statusJLabel = status;
        getPrimesJButton = getPrimes;
        cancelJButton = cancel;
        primes = new boolean[max];
    }

    // constructor

    // finds all primes up to max using the Sieve of Eratosthenes
    public Integer doInBackground() {
        Arrays.fill(primes, true);
    }

    // end constructor

    // finds all primes up to max using the Sieve of Eratosthenes
    public Integer doInBackground() {
        Arrays.fill(primes, true);
    }
```

Fig. 26.26 | Calculates the first \( n \) primes, displaying them as they are found. (Part 1 of 3.)
```java
int count = 0; // the number of primes found

// starting at the third value, cycle through the array and put
// false as the value of any greater number that is a multiple
for ( int i = 2; i < primes.length; i++ )
{
    if ( isCancelled() ) // if calculation has been canceled
        return count;
    else
    {
        setProgress( 100 * ( i + 1 ) / primes.length );
        try
        {
            Thread.sleep( generator.nextInt( 5 ) );
        } // end try
        catch ( InterruptedException ex )
        {
            statusJLabel.setText( "Worker thread interrupted" );
            return count;
        } // end catch
        if ( primes[ i ] ) // i is prime
        {
            publish( i ); // make i available for display in prime list
            ++count;
            for ( int j = i + i; j < primes.length; j += i )
                primes[ j ] = false; // i is not prime
        } // end else
    } // end else
} // end for

return count;
} // end method doInBackground

// displays published values in primes list
protected void process( List< Integer > publishedVals )
{
    for ( int i = 0; i < publishedVals.size(); i++ )
        intermediateTextArea.append( publishedVals.get( i ) + "\n" );
} // end method process

// code to execute when doInBackground completes
protected void done()
{
    getPrimesJButton.setEnabled( true ); // enable Get Primes button
    cancelJButton.setEnabled( false ); // disable Cancel button

    int numPrimes;
    try
    {

Fig. 26.26 | Calculates the first n primes, displaying them as they are found. (Part 2 of 3.)
Class `PrimeCalculator` extends `SwingWorker` (line 13), with the first type parameter indicating the return type of method `doInBackground` and the second indicating the type of intermediate results passed between methods `publish` and `process`. In this case, both type parameters are `Integer`s. The constructor (lines 23–34) takes as arguments an integer that indicates the upper limit of the prime numbers to locate, a `JTextArea` used to display primes in the GUI, one `JButton` for initiating a calculation and one for canceling it, and a `JLabel` used to display the status of the calculation.

**Sieve of Eratosthenes**

Line 33 initializes the elements of the boolean array `primes` to `true` with `Arrays` method `fill`. `PrimeCalculator` uses this array and the **Sieve of Eratosthenes** algorithm (described in Exercise 7.27) to find all primes less than `max`. The Sieve of Eratosthenes takes a list of integers and, beginning with the first prime number, filters out all multiples of that prime. It then moves to the next prime, which will be the next number that’s not yet filtered out, and eliminates all of its multiples. It continues until the end of the list is reached and all nonprimes have been filtered out. Algorithmically, we begin with element 2 of the boolean array and set the cells corresponding to all values that are multiples of 2 to `false` to indicate that they’re divisible by 2 and thus not prime. We then move to the next array element, check whether it’s `true`, and if so set all of its multiples to `false` to indicate that they’re divisible by the current index. When the whole array has been traversed in this way, all indices that contain `true` are prime, as they have no divisors.

**Method doInBackground**

In method `doInBackground` (lines 37–73), the control variable `i` for the loop (lines 43–70) controls the current index for implementing the Sieve of Eratosthenes. Line 45 calls the inherited `SwingWorker` method `isCancelled` to determine whether the user has
clicked the Cancel button. If isCancelled returns true, method doInBackground returns the number of primes found so far (line 46) without finishing the computation.

If the calculation isn’t canceled, line 49 calls setProgress to update the percentage of the array that’s been traversed so far. Line 55 puts the currently executing thread to sleep for up to 4 milliseconds. We discuss the reason for this shortly. Line 61 tests whether the element of array primes at the current index is true (and thus prime). If so, line 63 passes the index to method publish so that it can be displayed as an intermediate result in the GUI and line 64 increments the number of primes found. Lines 66–67 set all multiples of the current index to false to indicate that they’re not prime. When the entire array has been traversed, line 72 returns the number of primes found.

Method process
Lines 76–80 declare method process, which executes in the event dispatch thread and receives its argument publishedVals from method publish. The passing of values between publish in the worker thread and process in the event dispatch thread is asynchronous; process might not be invoked for every call to publish. All Integers published since the last call to process are received as a List by method process. Lines 78–79 iterate through this list and display the published values in a JTextArea. Because the computation in method doInBackground progresses quickly, publishing values often, updates to the JTextArea can pile up on the event dispatch thread, causing the GUI to become sluggish. In fact, when searching for a large number of primes, the event dispatch thread may receive so many requests in quick succession to update the JTextArea that it runs out of memory in its event queue. This is why we put the worker thread to sleep for a few milliseconds between calls to publish. The calculation is slowed just enough to allow the event dispatch thread to keep up with requests to update the JTextArea with new primes, enabling the GUI to update smoothly and remain responsive.

Method done
Lines 83–111 define method done. When the calculation is finished or canceled, method done enables the Get Primes button and disables the Cancel button (lines 85–86). Line 92 gets the return value—the number of primes found—from method doInBackground. Lines 94–108 catch the exceptions thrown by method get and display an appropriate message in the statusJLabel. If no exceptions occur, line 110 sets the statusJLabel to indicate the number of primes found.

Class FindPrimes
Class FindPrimes (Fig. 26.27) displays a JTextField that allows the user to enter a number, a JButton to begin finding all primes less than that number and a JTextArea to display the primes. A JButton allows the user to cancel the calculation, and a JProgressBar indicates the calculation’s progress. The FindPrimes constructor (lines 32–125) sets up the application’s GUI.

Lines 42–94 register the event handler for the getPrimesJButton. When the user clicks this JButton, lines 47–49 reset the JProgressBar and clear the displayPrimesJTextArea and the statusJLabel. Lines 53–63 parse the value in the JTextField and display an error message if the value is not an integer. Lines 66–68 construct a new PrimeCalculator object, passing as arguments the integer the user entered, the displayPrimesJTextArea for displaying the primes, the statusJLabel and the two JButtons.
public class FindPrimes extends JFrame {
    private final JTextField highestPrimeJTextField = new JTextField();
    private final JButton getPrimesJButton = new JButton("Get Primes");
    private final JTextArea displayPrimesJTextArea = new JTextArea();
    private final JButton cancelJButton = new JButton("Cancel");
    private final JProgressBar progressJProgressBar = new JProgressBar();
    private final JLabel statusJLabel = new JLabel();
    private PrimeCalculator calculator;

    public FindPrimes()
    {
        super( "Finding Primes with SwingWorker" );
        setLayout( new BorderLayout() );

        // initialize panel to get a number from the user
        JPanel northJPanel = new JPanel();
        northJPanel.add( new JLabel("Find primes less than: ") );
        highestPrimeJTextField.setColumns( 5 );
        northJPanel.add( highestPrimeJTextField );
        getPrimesJButton.addActionListener( new ActionListener()
        {
            public void actionPerformed( ActionEvent e )
            {
                progressJProgressBar.setValue( 0 ); // reset JProgressBar
                displayPrimesJTextArea.setText( "" ); // clear JTextArea
                statusJLabel.setText( "" ); // clear JLabel
            }
        } );
        int number; // search for primes up through this value
    }
}

Fig. 26.27  |  Using a SwingWorker to display prime numbers and update a JProgressBar while the prime numbers are being calculated. (Part 1 of 3.)
try {
    // get user input
    number = Integer.parseInt(
        highestPrimeJTextField.getText());
} // end try
catch ( NumberFormatException ex )
{
    statusJLabel.setText( "Enter an integer." );
    return;
} // end catch

// construct a new PrimeCalculator object
Calculator = new PrimeCalculator( number,
    displayPrimesJTextArea, statusJLabel, getPrimesJButton,
    cancelJButton );

// listen for progress bar property changes
Calculator.addPropertyChangeListener( new PropertyChangeListener() 

    public void propertyChange( PropertyChangeEvent e )
    {
        // if the changed property is progress,
        // update the progress bar
        if ( e.getPropertyName().equals( "progress" ) )
        {
            int newValue = ( Integer ) e.getNewValue();
            progressJProgressBar.setValue( newValue );
        }
    } // end method propertyChange
}; // end anonymous inner class

// disable Get Primes button and enable Cancel button
getPrimesJButton.setEnabled( false );
cancelJButton.setEnabled( true );

    calculator.execute(); // execute the PrimeCalculator object
} // end method ActionPerformed
); // end call to addActionListener
northJPanel.add( getPrimesJButton );

// add a scrollable JList to display results of calculation
displayPrimesJTextArea.setEditable( false );
add( new JScrollPane( displayPrimesJTextArea,
    JScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
    JScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER ) );

Fig. 26.27 | Using a SwingWorker to display prime numbers and update a JProgressBar while the prime numbers are being calculated. (Part 2 of 3.)
Chapter 26  Multithreading

Lines 71–85 register a `PropertyChangeListener` for the `PrimeCalculator` object. `PropertyChangeListener` is an interface from package `java.beans` that defines a single method, `propertyChange`. Every time method `setProgress` is invoked on a `PrimeCalculator`, `propertyChange` is called.

---

Fig. 26.27  | Using a `SwingWorker` to display prime numbers and update a `JProgressBar` while the prime numbers are being calculated. (Part 3 of 3.)
26.12 Interfaces Callable and Future

The PrimeCalculator generates a PropertyChangeEvent to indicate that the progress property has changed. Method propertyChange listens for these events. Line 78 tests whether a given PropertyChangeEvent indicates a change to the progress property. If so, line 80 gets the new value of the property and line 81 updates the JProgressBar with the new progress property value.

The Get Primes JButton is disabled (line 88) so only one calculation that updates the GUI can execute at a time, and the Cancel JButton is enabled (line 89) to allow the user to stop the computation before it completes. Line 91 executes the PrimesCalculator to begin finding primes. If the user clicks the cancelButton, the event handler registered at lines 107–115 calls PrimeCalculator’s method cancel (line 112), which is inherited from class SwingWorker, and the calculation returns early. The argument true to method cancel indicates that the thread performing the task should be interrupted in an attempt to cancel the task.

26.12 Interfaces Callable and Future

Interface Runnable provides only the most basic functionality for multithreaded programming. In fact, this interface has several limitations. Suppose a Runnable encounters a problem and tries to throw a checked exception. The run method is not declared to throw any exceptions, so the problem must be handled within the Runnable—the exception cannot be passed to the calling thread. Now suppose a Runnable is performing a long calculation and the application wants to retrieve the result of that calculation. The run method cannot return a value, so the application must use shared data to pass the value back to the calling thread. This also involves the overhead of synchronizing access to the data. The developers of the concurrency APIs recognized these limitations and created a new interface to fix them. The Callable interface (of package java.util.concurrent) declares a single method named call. This interface is designed to be similar to the Runnable interface—allowing an action to be performed concurrently in a separate thread—but the call method allows the thread to return a value or to throw a checked exception.

An application that creates a Callable likely wants to run it concurrently with other Runnables and Callables. The ExecutorService interface provides method submit, which will execute a Callable passed in as its argument. The submit method returns an object of type Future (of package java.util.concurrent), which is an interface that represents the executing Callable. The Future interface declares method get to return the result of the Callable and provides other methods to manage a Callable’s execution.

26.13 Java SE 7: Fork/Join Framework

Java SE 7’s concurrency APIs include the new fork/join framework, which helps programmers parallelize algorithms. The framework is beyond the scope of this book. Experts tell us that most Java programmers will benefit by this framework being used “behind the scenes” in the Java API and other third party libraries.

The fork/join framework is particularly well suited to divide-and-conquer-style algorithms, such as the merge sort that we implemented in Section 19.3.3. Recall that the recursive algorithm sorts an array by splitting it into two equal-sized subarrays, sorting each subarray, then merging them into one larger array. Each subarray is sorted by performing the same algorithm on the subarray. For algorithms like merge sort, the fork/join frame-
work can be used to create parallel tasks so that they can be distributed across multiple processors and be truly performed in parallel—the details of assigning the parallel tasks to different processors are handled for you by the framework.

To learn more about the fork/join framework and Java multithreading in general, please visit the sites listed in our Java Multithreading Resource Center at www.deitel.com/JavaMultithreading

26.14 Wrap-Up

In this chapter, you learned that concurrency has historically been implemented with operating-system primitives available only to experienced systems programmers, but that Java makes concurrency available to you through the language and APIs. You also learned that the JVM itself creates threads to run a program, and that it also can create threads to perform housekeeping tasks such as garbage collection.

We discussed the life cycle of a thread and the states that a thread may occupy during its lifetime. Next, we presented the interface Runnable, which is used to specify a task that can execute concurrently with other tasks. This interface’s run method is invoked by the thread executing the task. We showed how to execute a Runnable object by associating it with an object of class Thread. Then we showed how to use the Executor interface to manage the execution of Runnable objects via thread pools, which can reuse existing threads to eliminate the overhead of creating a new thread for each task and can improve performance by optimizing the number of threads to ensure that the processor stays busy.

You learned that when multiple threads share an object and one or more of them modify that object, indeterminate results may occur unless access to the shared object is managed properly. We showed you how to solve this problem via thread synchronization, which coordinates access to shared data by multiple concurrent threads. You learned several techniques for performing synchronization—first with the built-in class ArrayBlockingQueue (which handles all the synchronization details for you), then with Java’s built-in monitors and the synchronized keyword, and finally with interfaces Lock and Condition.

We discussed the fact that Swing GUIs are not thread safe, so all interactions with and modifications to the GUI must be performed in the event dispatch thread. We also discussed the problems associated with performing long-running calculations in the event dispatch thread. Then we showed how you can use the SwingWorker class to perform long-running calculations in worker threads. You learned how to display the results of a SwingWorker in a GUI when the calculation completed and how to display intermediate results while the calculation was still in process.

Finally, we discussed the Callable and Future interfaces, which enable you to execute tasks that return results and to obtain those results, respectively. We use the multithreading techniques introduced in this chapter again in Chapter 27, Networking, to help build multithreaded servers that can interact with multiple clients concurrently.

Summary

Section 26.1 Introduction

• Historically, concurrency (p. 1046) has been implemented with operating-system primitives available only to experienced systems programmers.
Summary

- The Ada programming language made concurrency primitives widely available.
- Java makes concurrency available to you through the language and APIs.
- The JVM creates threads to run a program and for housekeeping tasks such as garbage collection.

Section 26.2 Thread States: Life Cycle of a Thread
- A new thread begins its life cycle in the new state (p. 1048). When the program starts the thread, it’s placed in the runnable state. A thread in the runnable state is considered to be executing its task.
- A runnable thread transitions to the waiting state (p. 1048) to wait for another thread to perform a task. A waiting thread transitions to runnable when another thread notifies it to continue executing.
- A runnable thread can enter the timed waiting state (p. 1048) for a specified interval of time, transitioning back to runnable when that time interval expires or when the event it’s waiting for occurs.
- A runnable thread can transition to the timed waiting state if it provides an optional wait interval when it’s waiting for another thread to perform a task. Such a thread will return to the runnable state when it’s notified by another thread or when the timed interval expires.
- A sleeping thread (p. 1049) remains in the timed waiting state for a designated period of time, after which it returns to the runnable state.
- A runnable thread transitions to the blocked state (p. 1049) when it attempts to perform a task that cannot be completed immediately and the thread must temporarily wait until that task completes. At that point, the blocked thread transitions to the runnable state, so it can resume execution.
- A runnable thread enters the terminated state (p. 1049) when it successfully completes its task or otherwise terminates (perhaps due to an error).
- At the operating-system level, the runnable state (p. 1048) encompasses two separate states. When a thread first transitions to the runnable state from the new state, it’s in the ready state (p. 1049). A ready thread enters the running state (p. 1049) when the operating system dispatches it.
- Most operating systems allot a quantum (p. 1049) or timeslice in which a thread performs its task. When this expires, the thread returns to the ready state and another thread is assigned to the processor.
- Thread scheduling determines which thread to dispatch based on thread priorities.
- The job of an operating system’s thread scheduler (p. 1050) is to determine which thread runs next.
- When a higher-priority thread enters the ready state, the operating system generally preempts the currently running thread (an operation known as preemptive scheduling; p. 1050).
- Depending on the operating system, higher-priority threads could postpone—possibly indefinitely (p. 1050)—the execution of lower-priority threads.

Section 26.3 Creating and Executing Threads with Executor Framework
- A Runnable (p. 1051) object represents a task that can execute concurrently with other tasks.
- Interface Runnable declares method run (p. 1051) in which you place the code that defines the task to perform. The thread executing a Runnable calls method run to perform the task.
- A program will not terminate until its last thread completes execution.
- You cannot predict the order in which threads will be scheduled, even if you know the order in which they were created and started.
- It’s recommended that you use the Executor interface (p. 1051) to manage the execution of Runnable objects. An Executor object typically creates and manages a group of threads—called a thread pool (p. 1051).
- Executors (p. 1051) can reuse existing threads and can improve performance by optimizing the number of threads to ensure that the processor stays busy.
Chapter 26  Multithreading

• Executor method `execute` (p. 1051) receives a `Runnable` and assigns it to an available thread in a thread pool. If there are none, the Executor creates a new thread or waits for one to become available.

• Interface `ExecutorService` (of package `java.util.concurrent`; p. 1051) extends interface `Executor` and declares other methods for managing the life cycle of an Executor.

• An object that implements the `ExecutorService` interface can be created using static methods declared in class `Executors` (of package `java.util.concurrent`).

• `Executors` method `newCachedThreadPool` (p. 1052) returns an `ExecutorService` that creates new threads as they’re needed by the application.

• `ExecutorService` method `execute` executes its `Runnable` sometime in the future. The method returns immediately from each invocation—the program does not wait for each task to finish.

• `ExecutorService` method `shutdown` (p. 1054) notifies the `ExecutorService` to stop accepting new tasks, but continues executing existing tasks and terminates when those tasks complete execution.

Section 26.4 Thread Synchronization

• Thread synchronization (p. 1054) coordinates access to shared data by multiple concurrent threads.

• By synchronizing threads, you can ensure that each thread accessing a shared object excludes all other threads from doing so simultaneously—this is called mutual exclusion (p. 1054).

• A common way to perform synchronization is to use Java’s built-in monitors. Every object has a monitor and a monitor lock (p. 1055). The monitor ensures that its object’s monitor lock is held by a maximum of only one thread at any time, and thus can be used to enforce mutual exclusion.

• If an operation requires the executing thread to hold a lock while the operation is performed, a thread must acquire the lock (p. 1055) before it can proceed with the operation. Any other threads attempting to perform an operation that requires the same lock will be blocked until the first thread releases the lock, at which point the blocked threads may attempt to acquire the lock.

• To specify that a thread must hold a monitor lock to execute a block of code, the code should be placed in a `synchronized` statement (p. 1055). Such code is said to be guarded by the monitor lock (p. 1055).

• The `synchronized` statements are declared using the `synchronized` keyword:

   ```java
   synchronized (object) {
   statements
   } // end synchronized statement
   ```

   where `object` is the object whose monitor lock will be acquired; `object` is normally `this` if it’s the object in which the synchronized statement appears.

• Java also allows `synchronized` methods (p. 1055). Before executing, a non-static `synchronized` method must acquire the lock on the object that’s used to call the method. Similarly, a static `synchronized` method must acquire the lock on the class that’s used to call the method.

• `ExecutorService` method `awaitTermination` (p. 1058) forces a program to wait for threads to terminate. It returns control to its caller either when all tasks executing in the `ExecutorService` complete or when the specified timeout elapses. If all tasks complete before the timeout elapses, the method returns `true`; otherwise, it returns `false`.

• You can simulate atomicity (p. 1060) by ensuring that only one thread performs a set of operations at a time. Atomicity can be achieved with `synchronized` statements or `synchronized` methods.

• When you share immutable data across threads, you should declare the corresponding data fields `final` to indicate that variables’ values will not change after they’re initialized.
Section 26.5 Producer/Consumer Relationship without Synchronization
• In a multithreaded producer/consumer relationship (p. 1062), a producer thread generates data and places it in a shared object called a buffer. A consumer thread reads data from the buffer.
• Operations on a buffer data shared by a producer and a consumer should proceed only if the buffer is in the correct state. If the buffer is not full, the producer may produce; if the buffer is not empty, the consumer may consume. If the buffer is full when the producer attempts to write into it, the producer must wait until there’s space. If the buffer is empty or the previous value was already read, the consumer must wait for new data to become available.

Section 26.6 Producer/Consumer Relationship: ArrayBlockingQueue
• ArrayBlockingQueue (p. 1070) is a fully implemented buffer class from package java.util.concurrent that implements the BlockingQueue interface.
• An ArrayBlockingQueue can implement a shared buffer in a producer/consumer relationship. Method put (p. 1070) places an element at the end of the BlockingQueue, waiting if the queue is full. Method take (p. 1070) removes an element from the head of the BlockingQueue, waiting if the queue is empty.
• ArrayBlockingQueue stores shared data in an array that’s sized with an argument passed to the constructor. Once created, an ArrayBlockingQueue is fixed in size.

Section 26.7 Producer/Consumer Relationship with Synchronization
• You can implement a shared buffer yourself using the synchronized keyword and Object methods wait (p. 1073), notify and notifyAll.
• A thread can call Object method wait to release an object’s monitor lock, and wait in the waiting state while the other threads try to enter the object’s synchronized statement(s) or method(s).
• When a thread executing a synchronized statement (or method) completes or satisfies the condition on which another thread may be waiting, it can call Object method notify (p. 1073) to allow a waiting thread to transition to the runnable state. At this point, the thread that was transitioned can attempt to reacquire the monitor lock on the object.
• If a thread calls notifyAll (p. 1073), then all the threads waiting for the monitor lock become eligible to reacquire the lock (that is, they all transition to the runnable state).

Section 26.8 Producer/Consumer Relationship: Bounded Buffers
• You cannot make assumptions about the relative speeds of concurrent threads.
• A bounded buffer (p. 1080) can be used to minimize the amount of waiting time for threads that share resources and operate at the same average speeds. If the producer temporarily produces values faster than the consumer can consume them, the producer can write additional values into the extra buffer space (if any are available). If the consumer consumes faster than the producer produces new values, the consumer can read additional values (if there are any) from the buffer.
• The key to using a bounded buffer with a producer and consumer that operate at about the same speed is to provide the buffer with enough locations to handle the anticipated “extra” production.
• The simplest way to implement a bounded buffer is to use an ArrayBlockingQueue for the buffer so that all of the synchronization details are handled for you.

Section 26.9 Producer/Consumer Relationship: The Lock and Condition Interfaces
• The Lock and Condition interfaces (p. 1087) give programmers more precise control over thread synchronization, but are more complicated to use.
• Any object can contain a reference to an object that implements the Lock interface (of package java.util.concurrent.locks). A thread calls the Lock’s lock method (p. 1086) to acquire the
lock. Once a Lock has been obtained by one thread, the Lock object will not allow another thread to obtain the Lock until the first thread releases the Lock (by calling the Lock’s unlock method; p. 1086).

- If several threads are trying to call method lock on the same Lock object at the same time, only one thread can obtain the lock—the others are placed in the waiting state. When a thread calls unlock, the object’s lock is released and a waiting thread attempting to lock the object proceeds.
- Class ReentrantLock (p. 1087) is a basic implementation of the Lock interface.
- The ReentrantLock constructor takes a boolean that specifies whether the lock has a fairness policy (p. 1087). If true, the ReentrantLock’s fairness policy is “the longest-waiting thread will acquire the lock when it’s available”—this prevents indefinite postponement. If the argument is set to false, there’s no guarantee as to which waiting thread will acquire the lock when it’s available.
- If a thread that owns a Lock determines that it cannot continue with its task until some condition is satisfied, the thread can wait on a condition object (p. 1087). Using Lock objects allows you to explicitly declare the condition objects on which a thread may need to wait.
- Condition (p. 1087) objects are associated with a specific Lock and are created by calling Lock method newCondition, which returns a Condition object. To wait on a Condition, the thread can call the Condition’s await method. This immediately releases the associated Lock and places the thread in the waiting state for that Condition. Other threads can then try to obtain the Lock.
- When a runnable thread completes a task and determines that a waiting thread can now continue, the runnable thread can call Condition method signal to allow a thread in that Condition’s waiting state to return to the runnable state. At this point, the thread that transitioned from the waiting state to the runnable state can attempt to reacquire the Lock.
- If multiple threads are in a Condition’s waiting state when signal is called, the default implementation of Condition signals the longest-waiting thread to transition to the runnable state.
- If a thread calls Condition method signalAll, then all the threads waiting for that condition transition to the runnable state and become eligible to reacquire the Lock.
- When a thread is finished with a shared object, it must call method unlock to release the Lock.
- Locks allow you to interrupt waiting threads or to specify a timeout for waiting to acquire a lock—not possible with synchronized. Also, a Lock object is not constrained to be acquired and released in the same block of code, which is the case with the synchronized keyword.
- Condition objects allow you to specify multiple conditions on which threads may wait. Thus, it’s possible to indicate to waiting threads that a specific condition object is now true by calling that Condition object’s signal or signalAll methods (p. 1087). With synchronized, there’s no way to explicitly state the condition on which threads are waiting.

Section 26.11 Multithreading with GUI

- The event dispatch thread (p. 1095) handles interactions with the application’s GUI components. All tasks that interact with the GUI are placed in an event queue and executed sequentially by this thread.
- Swing GUI components are not thread safe. Thread safety is achieved by ensuring that Swing components are accessed from only the event dispatch thread.
- Performing a lengthy computation in response to a user interface interaction ties up the event dispatch thread, preventing it from attending to other tasks and causing the GUI components to become unresponsive. Long-running computations should be handled in separate threads.
- You can extend generic class SwingWorker (p. 1095; package javax.swing), which implements Runnable, to perform long-running computations in a worker thread and to update Swing components from the event dispatch thread based on the computations’ results. You override its
doInBackground and done methods. Method doInBackground performs the computation and returns the result. Method done displays the results in the GUI.

- Class SwingWorker’s first type parameter indicates the type returned by the doInBackground method; the second indicates the type that’s passed between the publish and process methods to handle intermediate results.
- Method doInBackground is called from a worker thread. After doInBackground returns, method done is called from the event dispatch thread to display the results.
- An ExecutionException is thrown if an exception occurs during the computation.
- SwingWorker method publish repeatedly sends intermediate results to method process, which displays the results in a GUI component. Method setProgress updates the progress property.
- Method process executes in the event dispatch thread and receives data from method publish. The passing of values between publish in the worker thread and process in the event dispatch thread is asynchronous; process is not necessarily invoked for every call to publish.
- PropertyChangeListener (p. 1108) is an interface from package java.beans that defines a single method, propertyChange. Every time method setProgress is invoked, a PropertyChangeEvent is generated to indicate that the progress property has changed.

Section 26.12 Interfaces Callable and Future
- The Callable (p. 1109) interface (of package java.util.concurrent) declares a single method named call that allows the thread to return a value or to throw a checked exception.
- ExecutorService method submit (p. 1109) executes a Callable passed in as its argument.
- Method submit returns an object of type Future (of package java.util.concurrent) that represents the executing Callable. Interface Future (p. 1109) declares method get to return the result of the Callable and provides other methods to manage a Callable’s execution.

Section 26.13 Java SE 7: Fork/Join Framework
- Java SE 7’s concurrency APIs include the new fork/join framework, which helps programmers parallelize algorithms. The fork/join framework particularly well suited to divide-and-conquer-style algorithms, like the merge sort.

Self-Review Exercises

26.1 Fill in the blanks in each of the following statements:

a) A thread enters the terminated state when ________.

b) To pause for a designated number of milliseconds and resume execution, a thread should call method ________ of class ________.

c) Method ________ of class Condition moves a single thread in an object’s waiting state to the runnable state.

d) Method ________ of class Condition moves every thread in an object’s waiting state to the runnable state.

e) A(n) ________ thread enters the ________ state when it completes its task or otherwise terminates.

f) A runnable thread can enter the ________ state for a specified interval of time.

g) At the operating-system level, the runnable state actually encompasses two separate states, ________ and ________.

h) Runnables are executed using a class that implements the ________ interface.

i) ExecutorService method ________ ends each thread in an ExecutorService as soon as it finishes executing its current Runnable, if any.
Chapter 26  Multithreading

j) A thread can call method _______ on a Condition object to release the associated Lock and place that thread in the _______ state.
k) In a(n) _______ relationship, the _______ generates data and stores it in a shared object, and the _______ reads data from the shared object.
l) Class _______ implements the BlockingQueue interface using an array.
m) Keyword _______ indicates that only one thread at a time should execute on an object.

26.2  State whether each of the following is true or false. If false, explain why.
a) A thread is not runnable if it has terminated.
b) Some operating systems use timeslicing with threads. Therefore, they can enable threads to preempt threads of the same priority.
c) When the thread’s quantum expires, the thread returns to the running state as the operating system assigns it to a processor.
d) On a single-processor system without timeslicing, each thread in a set of equal-priority threads (with no other threads present) runs to completion before other threads of equal priority get a chance to execute.

Answers to Self-Review Exercises

26.1  a) its run method ends. b) sleep, Thread. c) signal, signalAll. d) runnable, terminated. f) timed waiting, ready, running. h) Executor. i) await, waiting. k) producer/consumer, producer, consumer. l) ArrayBlockingQueue. m) synchronized.

26.2  a) True. b) False. Timeslicing allows a thread to execute until its timeslice (or quantum) expires. Then other threads of equal priority can execute. c) False. When a thread’s quantum expires, the thread returns to the ready state and the operating system assigns to the processor another thread. d) True.

Exercises

26.3  (True or False) State whether each of the following is true or false. If false, explain why.
a) Method sleep does not consume processor time while a thread sleeps.
b) Declaring a method synchronized guarantees that deadlock cannot occur.
c) Once a ReentrantLock has been obtained by a thread, the ReentrantLock object will not allow another thread to obtain the lock until the first thread releases it.
d) Swing components are thread safe.

26.4  (Multithreading Terms) Define each of the following terms.
a) thread  
b) multithreading  
c) runnable state  
d) timed waiting state  
e) preemptive scheduling  
f) Runnable interface  
g) notifyAll method  
h) producer/consumer relationship  
i) quantum

26.5  (Multithreading Terms) Discuss each of the following terms in the context of Java’s threading mechanisms.
a) synchronized  
b) producer  
c) consumer
d) wait
c) notify
e) Lock
f) Condition

d) List the reasons for entering the blocked state. For each of these, describe how the program will normally leave the blocked state and enter the runnable state.

26.7 (Deadlock and Indefinite Postponement) Two problems that can occur in systems that allow threads to wait are deadlock, in which one or more threads will wait forever for an event that cannot occur, and indefinite postponement, in which one or more threads will be delayed for some unpredictably long time. Give an example of how each of these problems can occur in multithreaded Java programs.

26.8 (Bouncing Ball) Write a program that bounces a blue ball inside a JPanel. The ball should begin moving with a mousePressed event. When the ball hits the edge of the JPanel, it should bounce off the edge and continue in the opposite direction. The ball should be updated using a Runnable.

26.9 (Bouncing Balls) Modify the program in Exercise 26.8 to add a new ball each time the user clicks the mouse. Provide for a minimum of 20 balls. Randomly choose the color for each new ball.

26.10 (Bouncing Balls with Shadows) Modify the program in Exercise 26.9 to add shadows. As a ball moves, draw a solid black oval at the bottom of the JPanel. You may consider adding a 3-D effect by increasing or decreasing the size of each ball when it hits the edge of the JPanel.

26.11 (Circular Buffer with Locks and Conditions) Reimplement the example in Section 26.8 using the Lock and Condition concepts presented in Section 26.9.