Java™

HOW TO PROGRAM

TENTH EDITION

EARLY OBJECTS

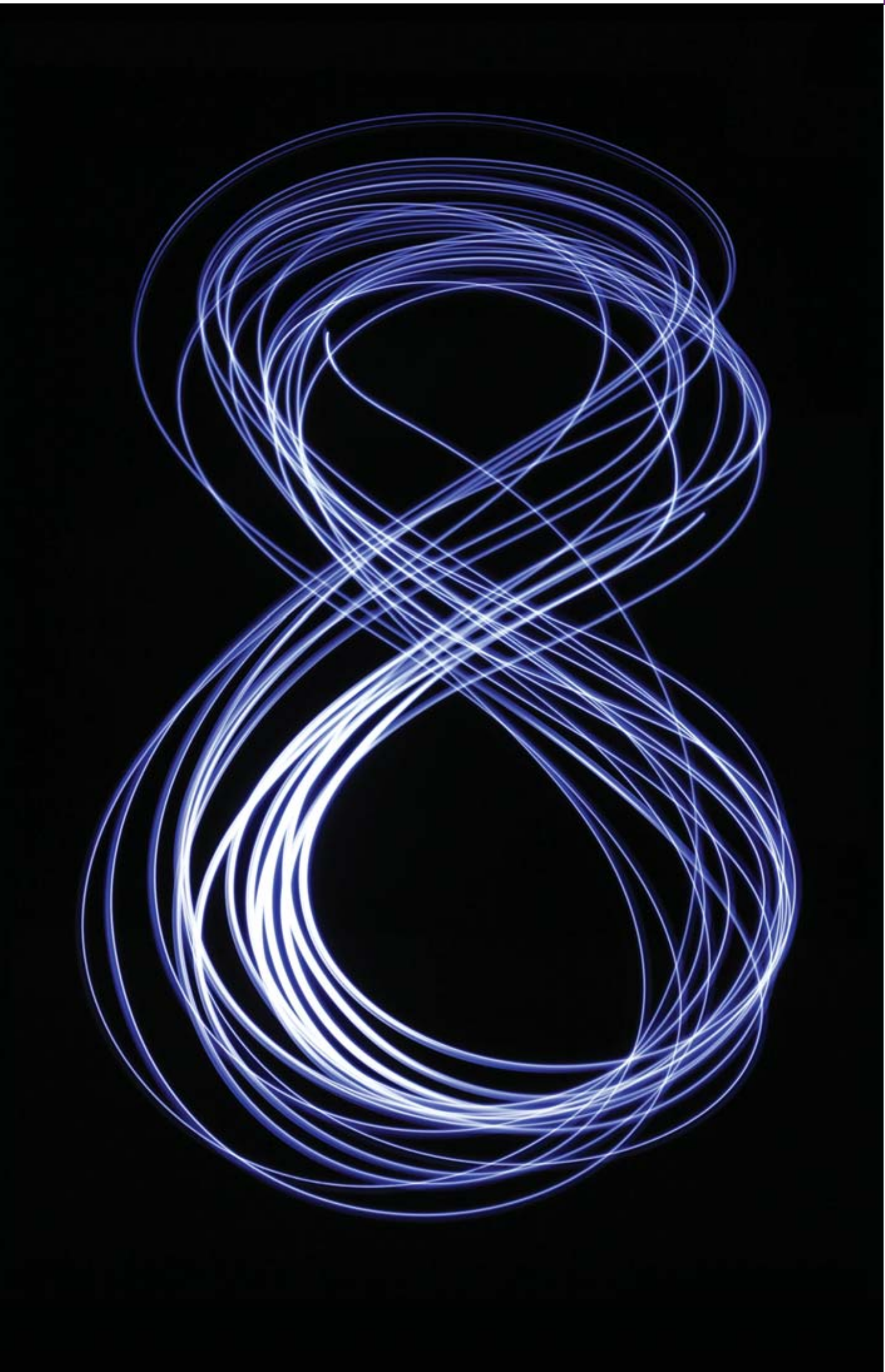
DEITEL®

PAUL DEITEL
HARVEY DEITEL

Use with
Java™ SE 7
or Java™ SE 8

Concurrency

23



The most general definition of beauty…Multeity in Unity.
—Samuel Taylor Coleridge

Do not block the way of inquiry.
—Charles Sanders Peirce

Learn to labor and to wait.
—Henry Wadsworth Longfellow

Objectives

In this chapter you'll:

- Understand concurrency, parallelism and multithreading.
- Learn the thread life cycle.
- Use `ExecutorService` to launch concurrent threads that execute `Runnable`s.
- Use `synchronized` methods to coordinate access to shared mutable data.
- Understand producer/consumer relationships.
- Use `SwingWorker` to update Swing GUIs in a thread-safe manner.
- Compare the performance of `Arrays` methods `sort` and `parallelSort` on a multi-core system.
- Use parallel streams for better performance on multi-core systems.
- Use `CompletableFuture`s to execute long calculations asynchronously and get the results in the future.

Outline

23.1 Introduction
23.2 Thread States and Life Cycle
- 23.2.1 *New* and *Runnable* States
- 23.2.2 *Waiting* State
- 23.2.3 *Timed Waiting* State
- 23.2.4 *Blocked* State
- 23.2.5 *Terminated* State
- 23.2.6 Operating-System View of the *Runnable* State
- 23.2.7 Thread Priorities and Thread Scheduling
- 23.2.8 Indefinite Postponement and Deadlock

23.3 Creating and Executing Threads with the `Executor` Framework
23.4 Thread Synchronization
- 23.4.1 Immutable Data
- 23.4.2 Monitors
- 23.4.3 Unsynchronized Mutable Data Sharing
- 23.4.4 Synchronized Mutable Data Sharing—Making Operations Atomic

23.5 Producer/Consumer Relationship without Synchronization
23.6 Producer/Consumer Relationship: `ArrayBlockingQueue`
23.7 (Advanced) Producer/Consumer Relationship with `synchronized`, `wait`, `notify` and `notifyAll`
23.8 (Advanced) Producer/Consumer Relationship: Bounded Buffers
23.9 (Advanced) Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces
23.10 Concurrent Collections
23.11 Multithreading with GUI: `SwingWorker`
- 23.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers
- 23.11.2 Processing Intermediate Results: Sieve of Eratosthenes

23.12 `sort`/`parallelSort` Timings with the Java SE 8 Date/Time API
23.13 Java SE 8: Sequential vs. Parallel Streams
23.14 (Advanced) Interfaces `Callable` and `Future`
23.15 (Advanced) Fork/Join Framework
23.16 Wrap-Up

Summary | *Self-Review Exercises* | *Answers to Self-Review Exercises* | *Exercises*

23.1 Introduction

[Note: Sections marked "Advanced" are intended for readers who wish a deeper treatment of concurrency and may be skipped by readers preferring only basic coverage.] It would be nice if we could focus our attention on performing only one task at a time and doing it well. That's usually difficult to do in a complex world in which there's so much going on at once. This chapter presents Java's capabilities for developing programs that create and manage multiple tasks. As we'll demonstrate, this can greatly improve program performance.

When we say that two tasks are operating **concurrently**, we mean that they're both *making progress* at once. Until recently, most computers had only a single processor. Operating systems on such computers execute tasks concurrently by rapidly switching between them, doing a small portion of each before moving on to the next, so that all tasks keep progressing. For example, it's common for personal computers to compile a program, send a file to a printer, receive electronic mail messages over a network and more, concurrently. Since its inception, Java has supported concurrency.

When we say that two tasks are operating **in parallel**, we mean that they're executing *simultaneously*. In this sense, parallelism is a subset of concurrency. The human body performs a great variety of operations in parallel. Respiration, blood circulation, digestion, thinking and walking, for example, can occur in parallel, as can all the senses—sight, hearing, touch, smell and taste. It's believed that this parallelism is possible because the human brain is thought to contain billions of "processors." Today's multi-core computers have multiple processors that can perform tasks in parallel.

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 and file handles. This capability is called **multithreading**.

> **Performance Tip 23.1**
>
> *A problem with single-threaded applications that can lead to poor responsiveness is that lengthy activities must complete* before *others can begin. In a multithreaded application, threads can be distributed across multiple cores (if available) so that multiple tasks execute in parallel and the application can operate more efficiently. Multithreading can also increase performance on single-processor systems—when one thread cannot proceed (because, for example, it's waiting for the result of an I/O operation), another can use the processor.*

Concurrent Programming Uses

We'll discuss many applications of **concurrent programming**. For example, when streaming an audio or video over the Internet, the user may not want to wait until the entire audio or video downloads before starting the playback. To solve this problem, multiple threads can be used—one to download the audio or video (later in the chapter we'll refer to this as a *producer*), and another to play it (later in the chapter we'll refer to this as a *consumer*). 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 audio or video in memory to keep the player thread busy. Producer and consumer threads *share memory*—we'll show how to coordinate these threads to ensure correct execution. The Java Virtual Machine (JVM) creates threads to run programs and threads to perform housekeeping tasks such as garbage collection.

Concurrent Programming Is Difficult

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!

Use the Prebuilt Classes of the Concurrency APIs Whenever Possible

Programming concurrent applications is difficult and error prone. If you must use synchronization in a program, follow these guidelines:

1. *The vast majority of programmers should use existing collection classes and interfaces from the concurrency APIs that manage synchronization for you—such as the* `Array-`

`BlockingQueue` class (an implementation of interface `BlockingQueue`) we discuss in Section 23.6. Two other concurrency API classes that you'll use frequently are `LinkedBlockingQueue` and `ConcurrentHashMap` (each summarized in Fig. 23.22). The concurrency API classes are written by experts, have been thoroughly tested and debugged, operate efficiently and help you avoid common traps and pitfalls. Section 23.10 overviews Java's pre-built concurrent collections.

2. For advanced programmers who want to control synchronization, use the `synchronized` keyword and `Object` methods `wait`, `notify` and `notifyAll`, which we discuss in the optional Section 23.7.
3. Only the most advanced programmers should use `Lock`s and `Condition`s, which we introduce in the optional Section 23.9, and classes like `LinkedTransferQueue`—an implementation of interface `TransferQueue`—which we summarize in Fig. 23.22.

You might want to read our discussions of the more advanced features in items 2 and 3 above, even though you most likely will not use them. We explain these because:

- They provide a solid basis for understanding how concurrent applications synchronize access to shared memory.
- By showing you the complexity involved in using these low-level features, we hope to impress upon you the message: *Use the simpler prebuilt concurrency capabilities whenever possible.*

23.2 Thread States and Life Cycle

At any time, a thread is said to be in one of several **thread states**—illustrated in the UML state diagram in Fig. 23.1. 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.

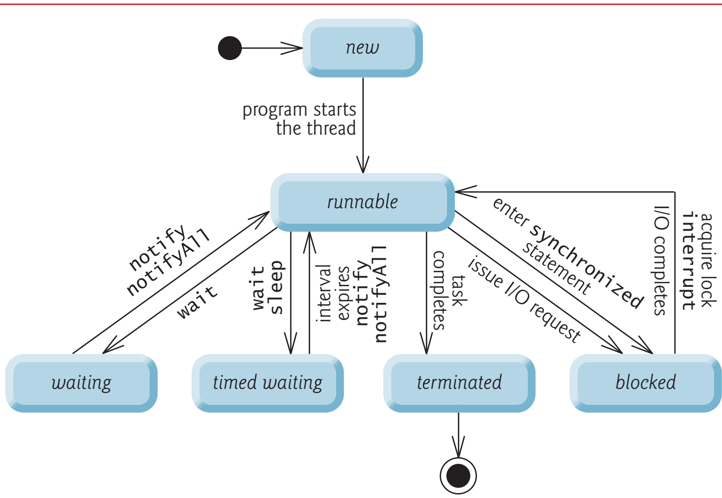


Fig. 23.1 | Thread life-cycle UML state diagram.

23.2.1 *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.

23.2.2 *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.

23.2.3 *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* threads 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.

23.2.4 *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.

23.2.5 *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. 23.1, the *terminated* state is followed by the UML final state (the bull's-eye symbol) to indicate the end of the state transitions.

23.2.6 Operating-System View of the *Runnable* State

At the operating system level, Java's *runnable* state typically encompasses *two separate* states (Fig. 23.2). 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.

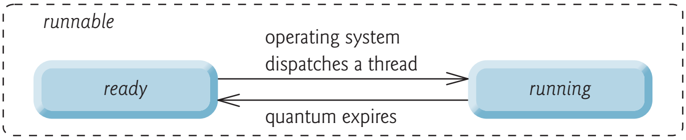


Fig. 23.2 | Operating system's internal view of Java's *runnable* state.

23.2.7 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 23.3).* 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.

Software Engineering Observation 23.1

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 23.1

Thread scheduling is platform dependent—the behavior of a multithreaded program could vary across different Java implementations.

23.2.8 Indefinite Postponement and Deadlock

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, a steady influx of 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 to ensure that the thread will eventually run.

Another problem related to indefinite postponement is called **deadlock**. This occurs when a waiting thread (let's call this thread1) cannot proceed because it's waiting (either directly or indirectly) for another thread (let's call this thread2) to proceed, while simultaneously thread2 cannot proceed because it's waiting (either directly or indirectly) for thread1 to proceed. The two threads are waiting for each other, so the actions that would enable each thread to continue execution can never occur.

23.3 Creating and Executing Threads with the 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`. It 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. `Executor`s 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 23.2

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`. You obtain an `ExecutorService` object by calling one of the `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. 23.3) implements `Runnable` (line 5), *so that multiple `PrintTask`s can execute concurrently*. Variable `sleepTime` (line 8) 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.

```
1  // Fig. 23.3: PrintTask.java
2  // PrintTask class sleeps for a random time from 0 to 5 seconds
3  import java.security.SecureRandom;
4
5  public class PrintTask implements Runnable
6  {
7     private static final SecureRandom generator = new SecureRandom();
8     private final int sleepTime; // random sleep time for thread
9     private final String taskName;
10
11    // constructor
12    public PrintTask(String taskName)
13    {
14       this.taskName = taskName;
15
16       // pick random sleep time between 0 and 5 seconds
17       sleepTime = generator.nextInt(5000); // milliseconds
18    }
19
20    // method run contains the code that a thread will execute
21    public void run()
22    {
23       try // put thread to sleep for sleepTime amount of time
24       {
25          System.out.printf("%s going to sleep for %d milliseconds.%n",
26             taskName, sleepTime);
```

Fig. 23.3 | `PrintTask` class sleeps for a random time from 0 to 5 seconds. (Part 1 of 2.)

```
27            Thread.sleep(sleepTime); // put thread to sleep
28         }
29         catch (InterruptedException exception)
30         {
31            exception.printStackTrace();
32            Thread.currentThread().interrupt(); // re-interrupt the thread
33         }
34
35         // print task name
36         System.out.printf("%s done sleeping%n", taskName);
37      }
38   } // end class PrintTask
```

Fig. 23.3 | `PrintTask` class sleeps for a random time from 0 to 5 seconds. (Part 2 of 2.)

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.

Let the Thread Handle `InterruptedException`*s*

It's considered good practice to let the executing thread handle `InterruptedException`s. Normally, you'd do this by declaring that method `run` throws the exception, rather than catching the exception. However, recall from Chapter 11 that when you override a method, the `throws` may contain only the same exception types or a subset of the exception types declared in the original method's `throws` clause. `Runnable` method `run` does not have a `throws` clause in its original declaration, so we cannot provide one in line 21. To ensure that the executing thread receives the `InterruptedException`, line 32 first obtains a reference to the currently executing `Thread` by calling `static` method **`currentThread`**, then uses that `Thread`'s `interrupt` method to deliver the `InterruptedException` to the current thread.[1]

Using the `ExecutorService` *to Manage Threads that Execute* `PrintTask`*s*

Figure 23.4 uses an `ExecutorService` object to manage threads that execute `PrintTask`s (as defined in Fig. 23.3). Lines 11–13 create and name three `PrintTask`s 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` to execute the `Runnable`s.

1. For detailed information on handling thread interruptions, see Chapter 7 of *Java Concurrency in Practice* by Brian Goetz, et al., Addison-Wesley Professional, 2006.

```java
// Fig. 23.4: TaskExecutor.java
// Using an ExecutorService to execute Runnables.
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

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 executorService = Executors.newCachedThreadPool();

      // start the three PrintTasks
      executorService.execute(task1); // start task1
      executorService.execute(task2); // start task2
      executorService.execute(task3); // start task3

      // shut down ExecutorService--it decides when to shut down threads
      executorService.shutdown();

      System.out.printf("Tasks started, main ends.%n%n");
   }
} // end class TaskExecutor
```

```
Starting Executor
Tasks started, main ends

task1 going to sleep for 4806 milliseconds
task2 going to sleep for 2513 milliseconds
task3 going to sleep for 1132 milliseconds
task3 done sleeping
task2 done sleeping
task1 done sleeping
```

```
Starting Executor
task1 going to sleep for 3161 milliseconds.
task3 going to sleep for 532 milliseconds.
task2 going to sleep for 3440 milliseconds.
Tasks started, main ends.

task3 done sleeping
task1 done sleeping
task2 done sleeping
```

Fig. 23.4 | Using an `ExecutorService` to execute `Runnable`s.

Lines 21–23 each invoke the `ExecutorService`'s `execute` method, which executes its `Runnable` 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 `Runnable`s have completed, the `ExecutorService` terminates. Line 28 outputs a message indicating that the tasks were started and the `main` thread is finishing its execution.

Main Thread

The code in `main` executes in the **main thread**, which is created by the JVM. The code in the `run` method of `PrintTask` (lines 21–37 of Fig. 23.3) executes whenever the `Executor` starts each `PrintTask`—again, this is sometime after they're passed to the `ExecutorService`'s `execute` method (Fig. 23.4, 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.

Sample Outputs

The sample outputs show each task's name and sleep time as the thread goes to sleep. The thread with the shortest sleep time *in most cases* awakens first, indicates that it's done sleeping and terminates. In Section 23.8, 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 `PrintTask`s output their names and sleep times. This shows that the `main` thread runs to completion before any of the `PrintTask`s gets a chance to run. In the second output, all of the `PrintTask`s output their names and sleep times *before* the `main` thread terminates. This shows that the `PrintTask`s 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*.

Waiting for Previously Scheduled Tasks to Terminate

After scheduling tasks to execute, you'll typically want to *wait for the tasks to complete*—for example, so that you can use the tasks' results. After calling method `shutdown`, you can call `ExecutorService` method `awaitTermination` to wait for scheduled tasks to complete. We demonstrate this in Fig. 23.7. We purposely did not call `awaitTermination` in Fig. 23.4 to demonstrate that a program can continue executing after the main thread terminates.

23.4 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 uncertain which thread's update takes effect. Similarly, if one thread is in the process of updating a shared object and another thread tries to read it, it's uncertain whether the reading thread will see the old value or the new one. In such cases, the program's behavior cannot be trusted—sometimes the program will produce the correct results, and sometimes it won't, and there won't be any 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 accesses the shared object. During that time, other threads desiring to access the object are kept waiting. When the thread with exclusive access finishes accessing the object, one of the waiting threads 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**.

23.4.1 Immutable Data

Actually, thread synchronization is necessary *only* for shared **mutable data**, i.e., data that may *change* during its lifetime. With shared **immutable data** that will *not* change, it's not possible for a thread to see old or incorrect values as a result of another thread's manipulation of 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, which could compromise thread safety. *Labeling object references as `final` indicates that the reference will not change, but it does not guarantee that the referenced object 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`.

Software Engineering Observation 23.3

Always declare data fields that you do not expect to change as `final`. Primitive variables that are declared as `final` can safely be shared across threads. An object reference that's declared as `final` ensures that the object it refers to will be fully constructed and initialized before it's used by the program, and prevents the reference from pointing to another object.

23.4.2 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 monitor allows only one thread at a time to execute statements within `synchronized` statements that lock on the same object, as only one thread at a time can hold the monitor lock. The `synchronized` statements are declared using the **`synchronized` keyword**:

```
synchronized (object)
{
    statements
}
```

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 in different threads 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 `synchronized` instance 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.

Software Engineering Observation 23.4

Using a `synchronized` block to enforce mutual exclusion is an example of the design pattern known as the Java Monitor Pattern (see section 4.2.1 of Java Concurrency in Practice *by Brian Goetz, et al., Addison-Wesley Professional, 2006).*

23.4.3 Unsynchronized Mutable Data Sharing

First, we illustrate the dangers of sharing an object across threads *without* proper synchronization. In this example (Figs. 23.5–23.7), two `Runnable`s 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. 23.5) will be *shared* across multiple threads. `SimpleArray` will enable those threads to place `int` values into `array` (declared at line 9). Line 10 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.

```
1   // Fig. 23.5: SimpleArray.java
2   // Class that manages an integer array to be shared by multiple threads.
3   import java.security.SecureRandom;
4   import java.util.Arrays;
5
6   public class SimpleArray // CAUTION: NOT THREAD SAFE!
7   {
8      private static final SecureRandom generator = new SecureRandom();
9      private final int[] array; // the shared integer array
10     private int writeIndex = 0; // shared index of next element to write
11
```

Fig. 23.5 | Class that manages an integer array to be shared by multiple threads. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 1 of 2.)

```
12      // construct a SimpleArray of a given size
13      public SimpleArray(int size)
14      {
15         array = new int[size];
16      }
17
18      // add a value to the shared array
19      public void add(int value)
20      {
21         int position = writeIndex; // store the write index
22
23         try
24         {
25            // put thread to sleep for 0-499 milliseconds
26            Thread.sleep(generator.nextInt(500));
27         }
28         catch (InterruptedException ex)
29         {
30            Thread.currentThread().interrupt(); // re-interrupt the thread
31         }
32
33         // put value in the appropriate element
34         array[position] = value;
35         System.out.printf("%s wrote %2d to element %d.%n",
36            Thread.currentThread().getName(), value, position);
37
38         ++writeIndex; // increment index of element to be written next
39         System.out.printf("Next write index: %d%n", writeIndex);
40      }
41
42      // used for outputting the contents of the shared integer array
43      public String toString()
44      {
45         return Arrays.toString(array);
46      }
47   } // end class SimpleArray
```

Fig. 23.5 | Class that manages an integer array to be shared by multiple threads. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 2 of 2.)

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 mutable 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. 23.6) 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 inserted in the `SimpleArray` object.

```java
1  // Fig. 23.6: ArrayWriter.java
2  // Adds integers to an array shared with other Runnables
3  import java.lang.Runnable;
4
5  public class ArrayWriter implements Runnable
6  {
7     private final SimpleArray sharedSimpleArray;
8     private final int startValue;
9
10    public ArrayWriter(int value, SimpleArray array)
11    {
12       startValue = value;
13       sharedSimpleArray = array;
14    }
15
16    public void run()
17    {
18       for (int i = startValue; i < startValue + 3; i++)
19       {
20          sharedSimpleArray.add(i); // add an element to the shared array
21       }
22    }
23 } // end class ArrayWriter
```

Fig. 23.6 | Adds integers to an array shared with other `Runnables`. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.)

Class `SharedArrayTest`

Class `SharedArrayTest` (Fig. 23.7) 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.

```java
1  // Fig. 23.7: SharedArrayTest.java
2  // Executing two Runnables to add elements to a shared SimpleArray.
3  import java.util.concurrent.Executors;
```

Fig. 23.7 | Executing two `Runnables` to add elements to a shared array. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 1 of 3.)

```java
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 executorService = Executors.newCachedThreadPool();
      executorService.execute(writer1);
      executorService.execute(writer2);

      executorService.shutdown();

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

         if (tasksEnded)
         {
            System.out.printf("%nContents of SimpleArray:%n");
            System.out.println(sharedSimpleArray); // print contents
         }
         else
            System.out.println(
               "Timed out while waiting for tasks to finish.");
      }
      catch (InterruptedException ex)
      {
         ex.printStackTrace();
      }
   } // end main
} // end class SharedArrayTest
```

```
pool-1-thread-1 wrote  1 to element 0.
Next write index: 1
pool-1-thread-1 wrote  2 to element 1.
Next write index: 2
pool-1-thread-1 wrote  3 to element 2.
Next write index: 3
pool-1-thread-2 wrote 11 to element 0.
Next write index: 4
```

First `pool-1-thread-1` wrote the value `1` to element `0`. Later `pool-1-thread-2` wrote the value `11` to element `0`, thus *overwriting* the previously stored value.

Fig. 23.7 | Executing two `Runnable`s to add elements to a shared array. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 2 of 3.)

```
pool-1-thread-2 wrote 12 to element 4.
Next write index: 5
pool-1-thread-2 wrote 13 to element 5.
Next write index: 6

Contents of SimpleArray:
[11, 2, 3, 0, 12, 13]
```

Fig. 23.7 | Executing two `Runnable`s to add elements to a shared array. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 3 of 3.)

`ExecutorService` *Method* `awaitTermination`

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`).

Method `awaitTermination` throws an `InterruptedException` if the calling thread is interrupted while waiting for other threads to terminate. Because we catch this exception in the application's `main` method, there's no need to re-interrupt the `main` thread as this program will terminate as soon as `main` terminates.

In this example, if *both* tasks complete before `awaitTermination` times out, line 34 displays the `SimpleArray` object's contents. Otherwise, lines 37–38 display a message indicating that the tasks did not finish executing before `awaitTermination` timed out.

Sample Program Output

Figure 23.7's output shows the problems (highlighted in the output) that can be caused by *failure to synchronize access to shared mutable 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 mutable 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 and unpredictably that a broken program does not produce incorrect results during testing, creating the illusion that the program is correct.* This is all the more reason to use predefined collections that handle the synchronization for you.

23.4.4 Synchronized Mutable Data Sharing—Making Operations Atomic

The output errors of Fig. 23.7 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`.

`SimpleArray` is *not thread safe because it allows any number of threads to read and modify shared mutable data concurrently*, which can cause errors. To make `SimpleArray` thread safe, we must ensure that no two threads can access its shared mutable data at the same time. 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. (As you'll see in later examples, read operations on shared mutable data should also be atomic.) 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 mutable 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 23.5

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 mutable data, hold the lock for the entirety of the operation to ensure that the operation is effectively atomic.

Class `SimpleArray` with Synchronization

Figure 23.8 displays class `SimpleArray` with the proper synchronization. Notice that it's identical to the `SimpleArray` class of Fig. 23.5, 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. 23.6) and `SharedArrayTest` (Fig. 23.7) from the previous example.

```java
// Fig. 23.8: SimpleArray.java
// Class that manages an integer array to be shared by multiple
// threads with synchronization.
import java.security.SecureRandom;
import java.util.Arrays;

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

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

   // add a value to the shared array
   public synchronized void add(int value)
   {
      int position = writeIndex; // store the write index

      try
      {
         // in real applications, you shouldn't sleep while holding a lock
         Thread.sleep(generator.nextInt(500)); // for demo only
      }
      catch (InterruptedException ex)
      {
         Thread.currentThread().interrupt();
      }

      // 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);
   }

   // used for outputting the contents of the shared integer array
   public synchronized String toString()
   {
      return Arrays.toString(array);
   }
} // end class SimpleArray
```

Fig. 23.8 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 1 of 2.)

```
pool-1-thread-1 wrote  1 to element 0.
Next write index: 1
pool-1-thread-2 wrote 11 to element 1.
Next write index: 2
pool-1-thread-2 wrote 12 to element 2.
Next write index: 3
pool-1-thread-2 wrote 13 to element 3.
Next write index: 4
pool-1-thread-1 wrote  2 to element 4.
Next write index: 5
pool-1-thread-1 wrote  3 to element 5.
Next write index: 6

Contents of SimpleArray:
[1, 11, 12, 13, 2, 3]
```

Fig. 23.8 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 2 of 2.)

Line 20 declares method `add` 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." [***Note:* Line 27 in this example calls `Thread` method `sleep` (for demo purposes only) to emphasize the unpredictability of thread scheduling. You should never call `sleep` while holding a lock in a real application.**]

Performance Tip 23.2

Keep the duration of `synchronized` statements as short as possible while maintaining the needed synchronization. This minimizes the wait time for blocked threads. Avoid performing I/O, lengthy calculations and operations that do not require synchronization while holding a lock.

23.5 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.

Examples of Producer/Consumer Relationship

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.

Synchronization and State Dependence

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 23.4. 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. Other examples of state dependence are that you can't drive your car if its gas tank is empty and you can't put more gas into the tank if it's already full.

Logic Errors from Lack of Synchronization

Consider how logic errors can arise if we do not synchronize access among multiple threads manipulating shared mutable data. Our next example (Figs. 23.9–23.13) 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. 23.12 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. The producer thread produces values from 1 through 10. 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. 23.13). 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.

Interface `Buffer`

The program consists of interface `Buffer` (Fig. 23.9) and classes `Producer` (Fig. 23.10), `Consumer` (Fig. 23.11), `UnsynchronizedBuffer` (Fig. 23.12) and `SharedBufferTest` (Fig. 23.13). Interface `Buffer` (Fig. 23.9) declares methods `blockingPut` (line 6) and `blockingGet` (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 `blockingPut` and `blockingGet` will call methods that throw `InterruptedExceptions`—typically this indicates that a method temporarily could be blocked from performing a task. We declare each method with a `throws` clause here so that we don't have to modify this interface for the later examples.

```
 1  // Fig. 23.9: Buffer.java
 2  // Buffer interface specifies methods called by Producer and Consumer.
 3  public interface Buffer
 4  {
 5     // place int value into Buffer
 6     public void blockingPut(int value) throws InterruptedException;
 7
 8     // return int value from Buffer
 9     public int blockingGet() throws InterruptedException;
10  } // end interface Buffer
```

Fig. 23.9 | Buffer interface specifies methods called by `Producer` and `Consumer`. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.)

Class `Producer`

Class `Producer` (Fig. 23.10) 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 15 of Fig. 23.13) 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` (Fig. 23.10, 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 `blockingPut` 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 36–37 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 `blockingPut`) executes as part of that task's thread of execution. This fact becomes important in Sections 23.6–23.8 when we add synchronization to the producer/consumer relationship.

```
// Fig. 23.10: Producer.java
// Producer with a run method that inserts the values 1 to 10 in buffer.
import java.security.SecureRandom;

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

   // constructor
   public Producer(Buffer sharedLocation)
   {
       this.sharedLocation = sharedLocation;
   }

   // store values from 1 to 10 in sharedLocation
   public void run()
   {
      int sum = 0;

      for (int count = 1; count <= 10; count++)
      {
         try // sleep 0 to 3 seconds, then place value in Buffer
         {
            Thread.sleep(generator.nextInt(3000)); // random sleep
            sharedLocation.blockingPut(count); // set value in buffer
            sum += count; // increment sum of values
            System.out.printf("\t%2d%n", sum);
         }
         catch (InterruptedException exception)
         {
            Thread.currentThread().interrupt();
         }
      }

      System.out.printf(
         "Producer done producing%nTerminating Producer%n");
   }
} // end class Producer
```

Fig. 23.10 | `Producer` with a `run` method that inserts the values 1 to 10 in buffer. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.)

Class `Consumer`

Class `Consumer` (Fig. 23.11) 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. 23.13) 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 21–34 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 `blockingGet` 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 36–37 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.

```java
1  // Fig. 23.11: Consumer.java
2  // Consumer with a run method that loops, reading 10 values from buffer.
3  import java.security.SecureRandom;
4
5  public class Consumer implements Runnable
6  {
7     private static final SecureRandom generator = new SecureRandom();
8     private final Buffer sharedLocation; // reference to shared object
9
10    // constructor
11    public Consumer(Buffer sharedLocation)
12    {
13       this.sharedLocation = sharedLocation;
14    }
15
16    // read sharedLocation's value 10 times and sum the values
17    public void run()
18    {
19       int sum = 0;
20
21       for (int count = 1; count <= 10; count++)
22       {
23          // sleep 0 to 3 seconds, read value from buffer and add to sum
24          try
25          {
26             Thread.sleep(generator.nextInt(3000));
27             sum += sharedLocation.blockingGet();
28             System.out.printf("\t\t\t%2d%n", sum);
29          }
30          catch (InterruptedException exception)
31          {
32             Thread.currentThread().interrupt();
33          }
34       }
35
36       System.out.printf("%n%s %d%n%s%n",
37          "Consumer read values totaling", sum, "Terminating Consumer");
38    }
39 } // end class Consumer
```

Fig. 23.11 | `Consumer` with a `run` method that loops, reading 10 values from buffer. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.)

We Call `Thread` *Method* `sleep` *Only for Demonstration Purposes*

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.

Class `UnsynchronizedBuffer` *Does Not Synchronize Access to the Buffer*

Class `UnsynchronizedBuffer` (Fig. 23.12) implements interface `Buffer` (line 4), but does *not* synchronize access to the buffer's state—we purposely do this to demonstrate the problems that occur when multiple threads access *shared mutable data* in *without* synchronization. Line 6 declares instance variable `buffer` and initializes it to `-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`. Again, methods `blockingPut` (lines 9–13) and `blockingGet` (lines 16–20) do *not* synchronize access to the `buffer` instance variable. Method `blockingPut` simply assigns its argument to `buffer` (line 12), and method `blockingGet` simply returns the value of `buffer` (line 19). As you'll see in Fig. 23.13, `UnsynchronizedBuffer` object is shared between the `Producer` and the `Consumer`.

```
1  // Fig. 23.12: UnsynchronizedBuffer.java
2  // UnsynchronizedBuffer maintains the shared integer that is accessed by
3  // a producer thread and a consumer thread.
4  public class UnsynchronizedBuffer implements Buffer
5  {
6     private int buffer = -1; // shared by producer and consumer threads
7
8     // place value into buffer
9     public void blockingPut(int value) throws InterruptedException
10    {
11       System.out.printf("Producer writes\t%2d", value);
12       buffer = value;
13    }
14
15    // return value from buffer
16    public int blockingGet() throws InterruptedException
17    {
18       System.out.printf("Consumer reads\t%2d", buffer);
19       return buffer;
20    }
21 } // end class UnsynchronizedBuffer
```

Fig. 23.12 | `UnsynchronizedBuffer` maintains the shared integer that is accessed by a producer thread and a consumer thread. (*Caution:* The example of Fig. 23.9–Fig. 23.13 is *not* thread safe.)

Class `SharedBufferTest`

In class `SharedBufferTest` (Fig. 23.13), line 12 creates an `ExecutorService` to execute the `Producer` and `Consumer` `Runnable`s. Line 15 creates an `UnsynchronizedBuffer` and assigns it to `Buffer` variable `sharedLocation`. This object stores the data that the `Producer` and `Consumer` threads will share. Lines 24–25 create and execute the `Producer` and `Consumer`. The `Producer` and `Consumer` constructors are each passed the same `Buffer` object (`sharedLocation`), so each object refers to the same `Buffer`. These lines also implicitly launch the threads and call each `Runnable`'s `run` method. Finally, line 27 calls method `shutdown` so that the application can terminate when the threads executing the `Producer` and `Consumer` complete their tasks and line 28 waits for the scheduled tasks to complete. When `main` terminates (line 29), the main thread of execution enters the *terminated* state.

```java
1  // Fig. 23.13: SharedBufferTest.java
2  // Application with two threads manipulating an unsynchronized buffer.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5  import java.util.concurrent.TimeUnit;
6
7  public class SharedBufferTest
8  {
9     public static void main(String[] args) throws InterruptedException
10    {
11       // create new thread pool with two threads
12       ExecutorService executorService = Executors.newCachedThreadPool();
13
14       // create UnsynchronizedBuffer to store ints
15       Buffer sharedLocation = new UnsynchronizedBuffer();
16
17       System.out.println(
18          "Action\t\tValue\tSum of Produced\tSum of Consumed");
19       System.out.printf(
20          "------\t\t-----\t---------------\t---------------%n%n");
21
22       // execute the Producer and Consumer, giving each
23       // access to the sharedLocation
24       executorService.execute(new Producer(sharedLocation));
25       executorService.execute(new Consumer(sharedLocation));
26
27       executorService.shutdown(); // terminate app when tasks complete
28       executorService.awaitTermination(1, TimeUnit.MINUTES);
29    }
30 } // end class SharedBufferTest
```

```
Action          Value   Sum of Produced Sum of Consumed
------          -----   --------------- ---------------

Producer writes  1       1
Producer writes  2       3                  — 1 is lost
Producer writes  3       6                  — 2 is lost
```

Fig. 23.13 | Application with two threads manipulating an unsynchronized buffer. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.) (Part 1 of 2.)

```
Consumer reads   3                          3
Producer writes  4       10
Consumer reads   4                          7
Producer writes  5       15
Producer writes  6       21                       5 is lost
Producer writes  7       28                       6 is lost
Consumer reads   7                         14
Consumer reads   7                         21     7 read again
Producer writes  8       36
Consumer reads   8                         29
Consumer reads   8                         37     8 read again
Producer writes  9       45
Producer writes 10       55                       9 is lost

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
```

```
Action          Value   Sum of Produced Sum of Consumed
------          -----   --------------- ---------------

Consumer reads  -1                         -1     reads -1 bad data
Producer writes  1        1
Consumer reads   1                          0
Consumer reads   1                          1     1 read again
Consumer reads   1                          2     1 read again
Consumer reads   1                          3     1 read again
Consumer reads   1                          4     1 read again
Producer writes  2        3
Consumer reads   2                          6
Producer writes  3        6
Consumer reads   3                          9
Producer writes  4       10
Consumer reads   4                         13
Producer writes  5       15
Producer writes  6       21                       5 is lost
Consumer reads   6                         19

Consumer read values totaling 19
Terminating Consumer
Producer writes  7       28                       7 never read
Producer writes  8       36                       8 never read
Producer writes  9       45                       9 never read
Producer writes 10       55                       10 never read

Producer done producing
Terminating Producer
```

Fig. 23.13 | Application with two threads manipulating an unsynchronized buffer. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.) (Part 2 of 2.)

Recall from this example's overview that the `Producer` should execute first and every value produced by the `Producer` should be consumed exactly once by the `Consumer`. However, when you study the first output of Fig. 23.13, notice 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 23.1

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 23.6 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.

23.6 Producer/Consumer Relationship: `ArrayBlockingQueue`

The best way to synchronize producer and consumer threads is to use classes from Java's `java.util.concurrent` package that *encapsulate the synchronization for you*. Java includes the class **`ArrayBlockingQueue`**—a fully implemented, *thread-safe buffer class* that implements interface **`BlockingQueue`**. This interface extends the `Queue` interface discussed in Chapter 16 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 mutable data in an array, the size of which is specified as an `ArrayBlockingQueue` constructor argument. Once created, an `ArrayBlockingQueue` is fixed in size and will not expand to accommodate extra elements.

Class `BlockingBuffer`

Figures 23.14–23.15 demonstrate a `Producer` and a `Consumer` accessing an `ArrayBlockingQueue`. Class `BlockingBuffer` (Fig. 23.14) 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 to mimic the `UnsynchronizedBuffer` example in Fig. 23.12. Lines 7 and 11 (Fig. 23.14) use generics, which we discussed in

Chapters 16–20. We discuss *multiple-element buffers* in Section 23.8. Because our `BlockingBuffer` class uses the *thread-safe* `ArrayBlockingQueue` class to manage all of its shared state (the shared buffer in this case), `BlockingBuffer` is itself *thread safe*, even though we have not implemented the synchronization ourselves.

```
1  // Fig. 23.14: BlockingBuffer.java
2  // Creating a synchronized buffer using an ArrayBlockingQueue.
3  import java.util.concurrent.ArrayBlockingQueue;
4
5  public class BlockingBuffer implements Buffer
6  {
7     private final ArrayBlockingQueue<Integer> buffer; // shared buffer
8
9     public BlockingBuffer()
10    {
11       buffer = new ArrayBlockingQueue<Integer>(1);
12    }
13
14    // place value into buffer
15    public void blockingPut(int value) throws InterruptedException
16    {
17       buffer.put(value); // place value in buffer
18       System.out.printf("%s%2d\t%s%d%n", "Producer writes ", value,
19          "Buffer cells occupied: ", buffer.size());
20    }
21
22    // return value from buffer
23    public int blockingGet() throws InterruptedException
24    {
25       int readValue = buffer.take(); // remove value from buffer
26       System.out.printf("%s %2d\t%s%d%n", "Consumer reads ",
27          readValue, "Buffer cells occupied: ", buffer.size());
28
29       return readValue;
30    }
31 } // end class BlockingBuffer
```

Fig. 23.14 | Creating a synchronized buffer using an `ArrayBlockingQueue`.

`BlockingBuffer` implements interface `Buffer` (Fig. 23.9) and uses classes `Producer` (Fig. 23.10 modified to remove line 28) and `Consumer` (Fig. 23.11 modified to remove line 28) from the example in Section 23.5. This approach demonstrates encapsulated synchronization—*the threads accessing the shared object are unaware that their buffer accesses are now synchronized*. The synchronization is handled entirely in the `blockingPut` and `blockingGet` 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 `blockingPut` and `blockingGet` methods.

Line 17 in method `blockingPut` (Fig. 23.14, 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 `blockingGet` (lines 23–30) calls the `ArrayBlocking-`

`Queue` 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`

Class `BlockingBufferTest` (Fig. 23.15) contains the `main` method that launches the application. Line 13 creates an `ExecutorService`, and line 16 creates a `BlockingBuffer` object and assigns its reference to the `Buffer` variable `sharedLocation`. Lines 18–19 execute the `Producer` and `Consumer` `Runnable`s. Line 21 calls method `shutdown` to end the application when the threads finish executing the `Producer` and `Consumer` tasks and line 22 waits for the scheduled tasks to complete.

```java
1  // Fig. 23.15: BlockingBufferTest.java
2  // Two threads manipulating a blocking buffer that properly
3  // implements the producer/consumer relationship.
4  import java.util.concurrent.ExecutorService;
5  import java.util.concurrent.Executors;
6  import java.util.concurrent.TimeUnit;
7
8  public class BlockingBufferTest
9  {
10    public static void main(String[] args) throws InterruptedException
11    {
12       // create new thread pool with two threads
13       ExecutorService executorService = Executors.newCachedThreadPool();
14
15       // create BlockingBuffer to store ints
16       Buffer sharedLocation = new BlockingBuffer();
17
18       executorService.execute(new Producer(sharedLocation));
19       executorService.execute(new Consumer(sharedLocation));
20
21       executorService.shutdown();
22       executorService.awaitTermination(1, TimeUnit.MINUTES);
23    }
24 } // end class BlockingBufferTest
```

```
Producer writes  1      Buffer cells occupied: 1
Consumer reads   1      Buffer cells occupied: 0
Producer writes  2      Buffer cells occupied: 1
Consumer reads   2      Buffer cells occupied: 0
Producer writes  3      Buffer cells occupied: 1
Consumer reads   3      Buffer cells occupied: 0
Producer writes  4      Buffer cells occupied: 1
Consumer reads   4      Buffer cells occupied: 0
Producer writes  5      Buffer cells occupied: 1
Consumer reads   5      Buffer cells occupied: 0
Producer writes  6      Buffer cells occupied: 1
```

Fig. 23.15 | Two threads manipulating a blocking buffer that properly implements the producer/consumer relationship. (Part 1 of 2.)

```
Consumer reads   6      Buffer cells occupied: 0
Producer writes  7      Buffer cells occupied: 1
Consumer reads   7      Buffer cells occupied: 0
Producer writes  8      Buffer cells occupied: 1
Consumer reads   8      Buffer cells occupied: 0
Producer writes  9      Buffer cells occupied: 1
Consumer reads   9      Buffer cells occupied: 0
Producer writes 10      Buffer cells occupied: 1

Producer done producing
Terminating Producer
Consumer reads  10      Buffer cells occupied: 0

Consumer read values totaling 55
Terminating Consumer
```

Fig. 23.15 | Two threads manipulating a blocking buffer that properly implements the producer/consumer relationship. (Part 2 of 2.)

While methods `put` and `take` of `ArrayBlockingQueue` are properly synchronized, `BlockingBuffer` methods `blockingPut` and `blockingGet` (Fig. 23.14) are not declared to be synchronized. Thus, the statements performed in method `blockingPut`—the `put` operation (line 17) and the output (lines 18–19)—are *not atomic*; nor are the statements in method `blockingGet`—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.

23.7 (Advanced) Producer/Consumer Relationship with synchronized, wait, notify and notifyAll

[*Note:* This section is intended for *advanced* programmers who want to control synchronization.[2]] 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 mutable 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` generally results in more-maintainable, better-performing code.*

After identifying the shared mutable data and the *synchronization policy* (i.e., associating the data with a lock that guards it), the next step in synchronizing access to the buffer is to implement methods `blockingGet` and `blockingPut` 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*.

2. For detailed information on `wait`, `notify` and `notifyAll`, see Chapter 14 of *Java Concurrency in Practice* by Brian Goetz, et al., Addison-Wesley Professional, 2006.

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` 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 23.1**
>
> *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`**.

> **Error-Prevention Tip 23.2**
>
> *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.*

Figures 23.16 and 23.17 demonstrate 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 23.5, except that line 28 is removed from class `Producer` and class `Consumer`.

Class `SynchronizedBuffer`

The synchronization is handled in class `SynchronizedBuffer`'s `blockingPut` and `blockingGet` methods (Fig. 23.16), which implements interface `Buffer` (line 4). Thus, the `Producer`'s and `Consumer`'s `run` methods simply call the shared object's `synchronized` `blockingPut` and blockingGet methods. Again, we output messages from this class's synchronized methods for demonstration purposes only—I/O *should not* be performed in synchronized blocks, because it's important to minimize the amount of time that an object is "locked."

```java
// Fig. 23.16: SynchronizedBuffer.java
// Synchronizing access to shared mutable 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;

   // place value into buffer
   public synchronized void blockingPut(int value)
      throws InterruptedException
   {
      // while there are no empty locations, place thread in waiting state
      while (occupied)
      {
         // output thread information and buffer information, then wait
         System.out.println("Producer tries to write."); // for demo only
         displayState("Buffer full. Producer waits."); // for demo only
         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); // for demo only

      notifyAll(); // tell waiting thread(s) to enter runnable state
   } // end method blockingPut; releases lock on SynchronizedBuffer

   // return value from buffer
   public synchronized int blockingGet() throws InterruptedException
   {
      // while no data to read, place thread in waiting state
      while (!occupied)
      {
         // output thread information and buffer information, then wait
         System.out.println("Consumer tries to read."); // for demo only
         displayState("Buffer empty. Consumer waits."); // for demo only
         wait();
      }

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

Fig. 23.16 | Synchronizing access to shared mutable data using Object methods wait and notifyAll. (Part 1 of 2.)

```
48
49          displayState("Consumer reads " + buffer); // for demo only
50
51          notifyAll(); // tell waiting thread(s) to enter runnable state
52
53          return buffer;
54       } // end method blockingGet; releases lock on SynchronizedBuffer
55
56       // display current operation and buffer state; for demo only
57       private synchronized void displayState(String operation)
58       {
59          System.out.printf("%-40s%d\t\t%b%n%n", operation, buffer,
60             occupied);
61       }
62    } // end class SynchronizedBuffer
```

Fig. 23.16 | Synchronizing access to shared mutable data using `Object` methods `wait` and `notifyAll`. (Part 2 of 2.)

Fields and Methods of Class `SynchronizedBuffer`

Class `SynchronizedBuffer` contains fields `buffer` (line 6) and `occupied` (line 7)—you must synchronize access to *both* fields to ensure that class `SynchronizedBuffer` is thread safe. Methods `blockingPut` (lines 10–31) and `blockingGet` (lines 34–54) 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 `blockingPut` and `blockingGet` 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 `blockingPut` *and the* `Producer` *Thread*

When the `Producer` thread's `run` method invokes `synchronized` method `blockingPut`, 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 14–20 first determines whether `occupied` is `true`. If so, `buffer` is *full* and we want to wait until the buffer is empty, so line 17 outputs a message indicating that the `Producer` thread is trying to write a value, and line 18 invokes method `displayState` (lines 57–61) to output another message indicating that `buffer` is *full* and that the `Producer` thread is *waiting* until there's space. Line 19 invokes method `wait` (inherited from `Object` by `SynchronizedBuffer`) to place the thread that called method `blockingPut` (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 `blockingPut` or `blockingGet` 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 `blockingPut` 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 `blockingPut` continues with the next statement after the loop.

Line 22 in method `blockingPut` assigns the `value` to the `buffer`. Line 26 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 28 invokes method `displayState` to output a message indicating that the `Producer` is writing a new value into the `buffer`. Line 30 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 `blockingPut` then returns to the caller (i.e., the `Producer`'s `run` method). When method `blockingPut` returns, it *implicitly releases the monitor lock* on the `SynchronizedBuffer` object.

Method `blockingGet` and the `Consumer` Thread

Methods `blockingGet` and `blockingPut` are implemented similarly. When the `Consumer` thread's `run` method invokes `synchronized` method `blockingGet`, 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 37–43 determines whether `occupied` is `false`. If so, the buffer is empty, so line 40 outputs a message indicating that the `Consumer` thread is trying to read a value, and line 41 invokes method `displayState` to output a message indicating that the buffer is *empty* and that the `Consumer` thread is *waiting*. Line 42 invokes method `wait` to place the thread that called method `blockingGet` (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 `blockingPut` or `blockingGet` method. If the lock on the `SynchronizedBuffer` is not available (e.g., if the `Producer` has not yet returned from method `blockingPut`), 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 `blockingGet` 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 `blockingGet` continues with the next statement after the loop. Line 47 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 49 calls method `displayState` to indicate that the consumer is reading and line 51 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 `blockingGet` returns the value of `buffer` to its caller. When method `blockingGet` returns, the lock on the `SynchronizedBuffer` object is *implicitly released*.

Error-Prevention Tip 23.3

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.*

Method `displayState` *Is Also* `synchronized`

Notice that method `displayState` is a `synchronized` method. This is important because it, too, reads the `SynchronizedBuffer`'s shared mutable data. Though only one thread at a time may acquire a given object's lock, one thread may acquire the same object's lock *multiple* times—this is known as a **reentrant lock** and enables one `synchronized` method to invoke another on the same object.

Testing Class `SynchronizedBuffer`

Class `SharedBufferTest2` (Fig. 23.17) is similar to class `SharedBufferTest` (Fig. 23.13). `SharedBufferTest2` contains method `main` (Fig. 23.17, lines 9–26), which launches the application. Line 12 creates an `ExecutorService` to run the `Producer` and `Consumer` tasks. Line 15 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 17–18 display the column heads for the output. Lines 21–22 execute a `Producer` and a `Consumer`. Finally, line 24 calls method `shutdown` to end the application when the `Producer` and `Consumer` complete their tasks and line 25 waits for the scheduled tasks to complete. When method `main` ends (line 26), the main thread of execution terminates.

```
1  // Fig. 23.17: SharedBufferTest2.java
2  // Two threads correctly manipulating a synchronized buffer.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5  import java.util.concurrent.TimeUnit;
6
7  public class SharedBufferTest2
8  {
9     public static void main(String[] args) throws InterruptedException
10    {
11       // create a newCachedThreadPool
12       ExecutorService executorService = Executors.newCachedThreadPool();
13
14       // create SynchronizedBuffer to store ints
15       Buffer sharedLocation = new SynchronizedBuffer();
16
17       System.out.printf("%-40s%s\t\t%s%n%-40s%s%n%n", "Operation",
18          "Buffer", "Occupied", "---------", "------\t\t--------");
19
```

Fig. 23.17 | Two threads correctly manipulating a synchronized buffer. (Part 1 of 3.)

```java
      // execute the Producer and Consumer tasks
      executorService.execute(new Producer(sharedLocation));
      executorService.execute(new Consumer(sharedLocation));

      executorService.shutdown();
      executorService.awaitTermination(1, TimeUnit.MINUTES);
   }
} // end class SharedBufferTest2
```

```
Operation                               Buffer          Occupied
---------                               ------          --------

Consumer tries to read.
Buffer empty. Consumer waits.           -1              false

Producer writes 1                       1               true

Consumer reads 1                        1               false

Consumer tries to read.
Buffer empty. Consumer waits.           1               false

Producer writes 2                       2               true

Consumer reads 2                        2               false

Producer writes 3                       3               true

Consumer reads 3                        3               false

Producer writes 4                       4               true

Producer tries to write.
Buffer full. Producer waits.            4               true

Consumer reads 4                        4               false

Producer writes 5                       5               true

Consumer reads 5                        5               false

Producer writes 6                       6               true

Producer tries to write.
Buffer full. Producer waits.            6               true

Consumer reads 6                        6               false

Producer writes 7                       7               true

Producer tries to write.
Buffer full. Producer waits.            7               true

Consumer reads 7                        7               false

Producer writes 8                       8               true

Consumer reads 8                        8               false

Consumer tries to read.
Buffer empty. Consumer waits.           8               false
```

Fig. 23.17 | Two threads correctly manipulating a synchronized buffer. (Part 2 of 3.)

```
Producer writes 9                                9               true

Consumer reads 9                                 9               false

Consumer tries to read.
Buffer empty. Consumer waits.                    9               false

Producer writes 10                               10              true

Consumer reads 10                                10              false


Producer done producing
Terminating Producer

Consumer read values totaling 55
Terminating Consumer
```

Fig. 23.17 | Two threads correctly manipulating a synchronized buffer. (Part 3 of 3.)

Study the outputs in Fig. 23.17. 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.

23.8 (Advanced) Producer/Consumer Relationship: Bounded Buffers

The program in Section 23.7 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 of them will spend more (or most) of its time waiting. For example, in the program in Section 23.7 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.

Performance Tip 23.3

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 23.6.) 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. An example of the producer/consumer relationship that uses a bounded buffer is video streaming, which we discussed in Section 23.1.

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. 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.

[*Note:* As we mention in Fig. 23.22, `ArrayBlockingQueue` can work with multiple producers and multiple consumers. For example, a factory that produces its product very fast will need to have many more delivery trucks (i.e., consumers) to remove those products quickly from the warehousing area (i.e., the bounded buffer) so that the factory can continue to produce products at full capacity.]

Performance Tip 23.4

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 23.6 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 23.13, we ask you to reimplement this section's example, using the Java Concurrency API techniques presented in Section 23.9.

Implementing Your Own Bounded Buffer as a Circular Buffer

The program in Figs. 23.18 and 23.19 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 23.5, except that line 28 is removed from class `Producer` and class `Consumer`. We implement the bounded buffer in class `CircularBuffer` (Fig. 23.18) 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. Once again, the output statements used in this class's `synchronized` methods are for *demonstration purposes only*.

```java
// Fig. 23.18: 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 blockingPut(int value)
      throws InterruptedException
   {
      // wait until buffer has space available, then write value;
      // while no empty locations, place thread in blocked state
      while (occupiedCells == buffer.length)
      {
         System.out.printf("Buffer is full. Producer waits.%n");
         wait(); // wait until a buffer cell is free
      } // end while

      buffer[writeIndex] = value; // set new buffer value

      // update circular write index
      writeIndex = (writeIndex + 1) % buffer.length;

      ++occupiedCells; // one more buffer cell is full
      displayState("Producer writes " + value);
      notifyAll(); // notify threads waiting to read from buffer
   }
```

Fig. 23.18 | Synchronizing access to a shared three-element bounded buffer. (Part 1 of 3.)

```java

   // return value from buffer
   public synchronized int blockingGet() throws InterruptedException
   {
      // wait until buffer has data, then read value;
      // while no data to read, place thread in waiting state
      while (occupiedCells == 0)
      {
         System.out.printf("Buffer is empty. Consumer waits.%n");
         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;
   }

   // display current operation and buffer state
   public synchronized void displayState(String operation)
   {
      // output operation and number of occupied buffer cells
      System.out.printf("%s%s%d)%n%s", operation,
         " (buffer cells occupied: ", occupiedCells, "buffer cells:  ");

      for (int value : buffer)
         System.out.printf(" %2d  ", value); // output values in buffer

      System.out.printf("%n               ");

      for (int i = 0; i < buffer.length; i++)
         System.out.print("---- ");

      System.out.printf("%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
      }

```

Fig. 23.18 | Synchronizing access to a shared three-element bounded buffer. (Part 2 of 3.)

```
85          System.out.printf("%n%n");
86       }
87    } // end class CircularBuffer
```

Fig. 23.18 | Synchronizing access to a shared three-element bounded buffer. (Part 3 of 3.)

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`, the circular buffer is *empty* 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`'s instance variables are *all* part of the class's shared mutable data, thus access to all of these variables must be synchronized to ensure that a `CircularBuffer` is thread safe.

`CircularBuffer` Method `blockingPut`

`CircularBuffer` method `blockingPut` (lines 12–31) performs the same tasks as in Fig. 23.16, with a few modifications. The loop at lines 17–21 determines whether the `Producer` must *wait* (i.e., all buffer cells are *full*). If so, line 19 indicates that the `Producer` is *waiting* to perform its task. Then line 20 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 23 after the `while` loop, the value written by the `Producer` is placed in the circular buffer at location `writeIndex`. Then line 26 updates `writeIndex` for the next call to `CircularBuffer` method `blockingPut`. 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 28 increments `occupiedCells`, because there's now one more value in the buffer that the `Consumer` can read. Next, line 29 invokes method `displayState` (lines 57–86) 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 30 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 `blockingGet`

`CircularBuffer` method `blockingGet` (lines 34–54) also performs the same tasks as it did in Fig. 23.16, with a few minor modifications. The loop at lines 38–42 (Fig. 23.18) determines whether the `Consumer` must wait (i.e., all buffer cells are *empty*). If the `Consumer` must *wait*, line 40 updates the output to indicate that the `Consumer` is *waiting* to perform its task. Then line 41 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 44 after a `notifyAll` call from the `Producer`, `readValue` is assigned the value at location `readIndex` in the circular buffer. Then line 47 updates `readIndex` for the next call to `CircularBuffer` method `blockingGet`. This line and line 26 implement the *circularity* of the buffer. Line 49 decrements `occupiedCells`, because there's now one more position in the buffer in which the `Producer` thread can place a value. Line 50 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 51 invokes method `notifyAll` to allow any `Producer` threads *waiting to write* into the `CircularBuffer` object to attempt to write again. Then line 53 returns the consumed value to the caller.

`CircularBuffer` *Method* `displayState`

Method `displayState` (lines 57–86) outputs the application's state. Lines 63–64 output the values of the buffer cells. Line 64 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 71–83 output the current `writeIndex` and `readIndex` with the letters `W` and `R`, respectively. Once again, `displayState` is a synchronized method because it accesses class `CircularBuffer`'s shared mutable data.

Testing Class `CircularBuffer`

Class `CircularBufferTest` (Fig. 23.19) contains the `main` method that launches the application. Line 12 creates the `ExecutorService`, and line 15 creates a `CircularBuffer` object and assigns its reference to `CircularBuffer` variable `sharedLocation`. Line 18 invokes the `CircularBuffer`'s `displayState` method to show the initial state of the buffer. Lines 21–22 execute the `Producer` and `Consumer` tasks. Line 24 calls method `shutdown` to end the application when the threads complete the `Producer` and `Consumer` tasks and line 25 waits for the tasks to complete.

```java
1  // Fig. 23.19: CircularBufferTest.java
2  // Producer and Consumer threads correctly manipulating a circular buffer.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5  import java.util.concurrent.TimeUnit;
6
7  public class CircularBufferTest
8  {
9     public static void main(String[] args) throws InterruptedException
10    {
11       // create new thread pool with two threads
12       ExecutorService executorService = Executors.newCachedThreadPool();
13
14       // create CircularBuffer to store ints
15       CircularBuffer sharedLocation = new CircularBuffer();
16
17       // display the initial state of the CircularBuffer
18       sharedLocation.displayState("Initial State");
19
20       // execute the Producer and Consumer tasks
21       executorService.execute(new Producer(sharedLocation));
22       executorService.execute(new Consumer(sharedLocation));
23
24       executorService.shutdown();
25       executorService.awaitTermination(1, TimeUnit.MINUTES);
26    }
27 } // end class CircularBufferTest
```

Fig. 23.19 | `Producer` and `Consumer` threads correctly manipulating a circular buffer. (Part 1 of 3.)

```
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

Consumer reads 1 (buffer cells occupied: 0)
buffer cells:    1   -1   -1
               ---- ---- ----
                     WR

Buffer is empty. Consumer waits.
Producer writes 2 (buffer cells occupied: 1)
buffer cells:    1    2   -1
               ---- ---- ----
                      R   W

Consumer reads 2 (buffer cells occupied: 0)
buffer cells:    1    2   -1
               ---- ---- ----
                          WR

Producer writes 3 (buffer cells occupied: 1)
buffer cells:    1    2    3
               ---- ---- ----
                W          R

Consumer reads 3 (buffer cells occupied: 0)
buffer cells:    1    2    3
               ---- ---- ----
                WR

Producer writes 4 (buffer cells occupied: 1)
buffer cells:    4    2    3
               ---- ---- ----
                 R   W

Producer writes 5 (buffer cells occupied: 2)
buffer cells:    4    5    3
               ---- ---- ----
                 R        W

Consumer reads 4 (buffer cells occupied: 1)
buffer cells:    4    5    3
               ---- ---- ----
                      R   W

Producer writes 6 (buffer cells occupied: 2)
buffer cells:    4    5    6
               ---- ---- ----
                W     R
```

Fig. 23.19 | Producer and Consumer threads correctly manipulating a circular buffer. (Part 2 of 3.)

```
Producer writes 7 (buffer cells occupied: 3)
buffer cells:    7    5    6
               ---- ---- ----
                     WR

Consumer reads 5 (buffer cells occupied: 2)
buffer cells:    7    5    6
                ---- ---- ----
                     W     R

Producer writes 8 (buffer cells occupied: 3)
buffer cells:    7    8    6
               ---- ---- ----
                          WR

Consumer reads 6 (buffer cells occupied: 2)
buffer cells:    7    8    6
               ---- ---- ----
                 R        W

Consumer reads 7 (buffer cells occupied: 1)
buffer cells:    7    8    6
               ---- ---- ----
                      R   W

Producer writes 9 (buffer cells occupied: 2)
buffer cells:    7    8    9
               ---- ---- ----
                W     R

Consumer reads 8 (buffer cells occupied: 1)
buffer cells:    7    8    9
               ---- ---- ----
                W          R

Consumer reads 9 (buffer cells occupied: 0)
buffer cells:    7    8    9
               ---- ---- ----
                WR

Producer writes 10 (buffer cells occupied: 1)
buffer cells:   10    8    9
               ---- ---- ----
                 R   W


Producer done producing
Terminating Producer
Consumer reads 10 (buffer cells occupied: 0)
buffer cells:   10    8    9
               ---- ---- ----
                     WR


Consumer read values totaling: 55
Terminating Consumer
```

Fig. 23.19 | `Producer` and `Consumer` threads correctly manipulating a circular buffer. (Part 3 of 3.)

Each time the `Producer` writes a value or the `Consumer` reads a value, the program outputs a message indicating the action performed (a read or a write), the contents of `buffer`, and the location of `writeIndex` and `readIndex`. In the output of Fig. 23.19, the `Producer` first writes the value `1`. The buffer then contains the value `1` in the first cell and the value `-1` (the default value that we use for output purposes) in the other two cells. The write index is updated to the second cell, while the read index stays at the first cell. Next, the `Consumer` reads `1`. The buffer contains the same values, but the read index has been updated to the second cell. The `Consumer` then tries to read again, but the buffer is empty and the `Consumer` is forced to wait. Only once in this execution of the program was it necessary for either thread to wait.

23.9 (Advanced) 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. *Only the most advanced programmers should use these interfaces.*

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 exiting 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.

Error-Prevention Tip 23.4

Place calls to `Lock` method `unlock` in a `finally` block. If an exception is thrown, `unlock` must still be called or deadlock could occur.

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 23.6

Using a `ReentrantLock` with a fairness policy avoids indefinite postponement.

Performance Tip 23.5

In most cases, a non-fair lock is preferable, because using a fair lock 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`.

Error-Prevention Tip 23.5

When multiple threads manipulate a shared object using locks, ensure that if one thread calls method `await` to enter the waiting *state for a condition object, a separate thread eventually will call `Condition` method `signal` to transition the thread waiting on the condition object back to the* runnable *state. If multiple threads may be waiting on the condition object, a separate thread can call `Condition` method `signalAll` as a safeguard to ensure that all the waiting threads have another opportunity to perform their tasks. If this is not done, starvation might occur.*

Common Programming Error 23.2

An `IllegalMonitorStateException` occurs if a thread issues an `await`, a `signal`, or a `signalAll` on a `Condition` object that was created from a `ReentrantLock` without having acquired the lock for that `Condition` object.

`Lock` and `Condition` vs. the `synchronized` Keyword

In some applications, using `Lock` and `Condition` objects may be preferable to using the `synchronized` keyword. `Lock`s 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.

Software Engineering Observation 23.7

Think of `Lock` and `Condition` as an advanced version of `synchronized`. `Lock` and `Condition` support timed waits, interruptible waits and multiple `Condition` queues per `Lock`—if you do not need one of these features, you do not need `Lock` and `Condition`.

Error-Prevention Tip 23.6

Using interfaces `Lock` and `Condition` is error prone—`unlock` is not guaranteed to be called, whereas the monitor in a `synchronized` statement will always be released when the statement completes execution. Of course, you can guarantee that `unlock` will be called if it's placed in a `finally` block, as we do in Fig. 23.20.

Using `Lock`s and `Condition`s to Implement Synchronization

We now implement the producer/consumer relationship using `Lock` and `Condition` objects to coordinate access to a shared single-element buffer (Figs. 23.20 and 23.21). 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 23.5, except that line 28 is removed from class `Producer` and class `Consumer`.

Class `SynchronizedBuffer`

Class `SynchronizedBuffer` (Fig. 23.20) 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 `Condition`s 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 mutable data. The `boolean` variable `occupied` (line 18) keeps track of whether the buffer currently holds data (that the `Consumer` should read).

```java
// Fig. 23.20: 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
{
   // Lock to control synchronization with this buffer
   private final Lock accessLock = new ReentrantLock();

   // conditions to control reading and writing
   private final Condition canWrite = accessLock.newCondition();
   private final Condition canRead = accessLock.newCondition();

   private int buffer = -1; // shared by producer and consumer threads
   private boolean occupied = false; // whether buffer is occupied

   // place int value into buffer
   public void blockingPut(int value) throws InterruptedException
   {
      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
         }

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

Fig. 23.20 | Synchronizing access to a shared integer using the Lock and Condition interfaces. (Part 1 of 2.)

```java

    // return value from buffer
    public int blockingGet() 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
          }

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

       return readValue;
    }

    // display current operation and buffer state
    private void displayState(String operation)
    {
       try
       {
          accessLock.lock(); // lock this object
          System.out.printf("%-40s%d\t\t%b%n%n", operation, buffer,
             occupied);
       }
       finally
       {
          accessLock.unlock(); // unlock this objects
       }
    }
 } // end class SynchronizedBuffer
```

Fig. 23.20 | Synchronizing access to a shared integer using the `Lock` and `Condition` interfaces. (Part 2 of 2.)

Line 23 in method `blockingPut` 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 `blockingGet` (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 `blockingPut` method (line 45). When the `Condition` object is *signaled*, the `blockingGet` method continues. Lines 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.

Common Programming Error 23.3

Forgetting to `signal` a waiting thread is a logic error. The thread will remain in the waiting *state, which will prevent it from proceeding. Such waiting can lead to indefinite postponement or deadlock.*

Class `SharedBufferTest2`

Class `SharedBufferTest2` (Fig. 23.21) is identical to that of Fig. 23.17. Study the outputs in Fig. 23.21. *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. 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.

```java
// Fig. 23.21: SharedBufferTest2.java
// Two threads manipulating a synchronized buffer.
import java.util.concurrent.ExecutorService;
```

Fig. 23.21 | Two threads manipulating a synchronized buffer. (Part 1 of 3.)

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

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

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

      System.out.printf("%-40s%s\t\t%s%n%-40s%s%n%n", "Operation",
         "Buffer", "Occupied", "---------", "------\t\t--------");

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

      executorService.shutdown();
      executorService.awaitTermination(1, TimeUnit.MINUTES);
   }
} // end class SharedBufferTest2
```

```
Operation                               Buffer          Occupied
---------                               ------          --------

Producer writes 1                       1               true

Producer tries to write.
Buffer full. Producer waits.            1               true

Consumer reads 1                        1               false

Producer writes 2                       2               true

Producer tries to write.
Buffer full. Producer waits.            2               true

Consumer reads 2                        2               false

Producer writes 3                       3               true

Consumer reads 3                        3               false

Producer writes 4                       4               true

Consumer reads 4                        4               false

Consumer tries to read.
Buffer empty. Consumer waits.           4               false
```

Fig. 23.21 | Two threads manipulating a synchronized buffer. (Part 2 of 3.)

```
Producer writes 5                       5               true

Consumer reads 5                        5               false

Consumer tries to read.
Buffer empty. Consumer waits.           5               false

Producer writes 6                       6               true

Consumer reads 6                        6               false

Producer writes 7                       7               true

Consumer reads 7                        7               false

Producer writes 8                       8               true

Consumer reads 8                        8               false

Producer writes 9                       9               true

Consumer reads 9                        9               false

Producer writes 10                      10              true


Producer done producing
Terminating Producer
Consumer reads 10                       10              false


Consumer read values totaling 55
Terminating Consumer
```

Fig. 23.21 | Two threads manipulating a synchronized buffer. (Part 3 of 3.)

23.10 Concurrent Collections

In Chapter 16, 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 sharing collections among multiple threads.

Figure 23.22 lists the many concurrent collections in package `java.util.concurrent`. The entries for `ConcurrentHashMap` and `LinkedBlockingQueue` are shown in bold because these are by far the most frequently used concurrent collections. Like the collections introduced in Chapter 16, the concurrent collections have been enhanced to support lambdas. However, rather than providing methods to support streams, the concurrent collections provide their own implementations of various stream-like operations—e.g., `ConcurrentHashMap` has methods `forEach`, `reduce` and `search`—that are designed and optimized for concurrent collections that are shared among threads. For more information on the concurrent collections, visit

Java SE 7:
`http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`

Java SE 8
`http://download.java.net/jdk8/docs/api/java/util/concurrent/package-summary.html`

Collection	Description
`ArrayBlockingQueue`	A fixed-size queue that supports the producer/consumer relationship—possibly with many producers and consumers.
`ConcurrentHashMap`	**A hash-based map (similar to the `HashMap` introduced in Chapter 16) that allows an arbitrary number of reader threads and a limited number of writer threads. This and the `LinkedBlockingQueue` are by far the most frequently used concurrent collections.**
`ConcurrentLinkedDeque`	A concurrent linked-list implementation of a double-ended queue.
`ConcurrentLinkedQueue`	A concurrent linked-list implementation of a queue that can grow dynamically.
`ConcurrentSkipListMap`	A concurrent map that is sorted by its keys.
`ConcurrentSkipListSet`	A sorted concurrent set.
`CopyOnWriteArrayList`	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.
`CopyOnWriteArraySet`	A set that's implemented using `CopyOnWriteArrayList`.
`DelayQueue`	A variable-size queue containing `Delayed` objects. An object can be removed only after its delay has expired.
`LinkedBlockingDeque`	A double-ended blocking queue implemented as a linked list that can optionally be fixed in size.
`LinkedBlockingQueue`	**A blocking queue implemented as a linked list that can optionally be fixed in size. This and the `ConcurrentHashMap` are by far the most frequently used concurrent collections.**
`LinkedTransferQueue`	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.
`PriorityBlockingQueue`	A variable-length priority-based blocking queue (like a `PriorityQueue`).
`SynchronousQueue`	[For experts.] 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.

Fig. 23.22 | Concurrent collections summary (package `java.util.concurrent`).

23.11 Multithreading with GUI: SwingWorker

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 that might corrupt the GUI. 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 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.

It's acceptable to perform brief 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 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, you must update the GUI with the computation's results from the event dispatch thread, rather than from the worker thread that performed the computation.

Class SwingWorker

Class **SwingWorker** (in package javax.swing) enables you to perform an asynchronous task in a worker thread (such as a long-running computation) then update Swing components from the event dispatch thread based on the task's 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 a task in a worker thread and making its results available for display in a GUI. Some common SwingWorker methods are described in Fig. 23.23.

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

Fig. 23.23 | Commonly used SwingWorker methods. (Part 1 of 2.)

Method	Description
`process`	Receives intermediate results from the `publish` method and processes these results on the event dispatch thread.
`setProgress`	Sets the progress property to notify any property change listeners on the event dispatch thread of progress bar updates.

Fig. 23.23 | Commonly used `SwingWorker` methods. (Part 2 of 2.)

23.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers

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 18.5. 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 `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. 23.24) performs the recursive Fibonacci calculation in a *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. 23.24: 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
8  public class BackgroundCalculator extends SwingWorker<Long, Object>
9  {
10    private final int n; // Fibonacci number to calculate
11    private final JLabel resultJLabel; // JLabel to display the result
12
13    // constructor
14    public BackgroundCalculator(int n, JLabel resultJLabel)
15    {
16       this.n = n;
17       this.resultJLabel = resultJLabel;
18    }
```

Fig. 23.24 | `SwingWorker` subclass for calculating Fibonacci numbers in a background thread. (Part 1 of 2.)

```
19
20      // long-running code to be run in a worker thread
21      public Long doInBackground()
22      {
23         return nthFib = fibonacci(n);
24      }
25
26      // code to run on the event dispatch thread when doInBackground returns
27      protected void done()
28      {
29         try
30         {
31            // get the result of doInBackground and display it
32            resultJLabel.setText(get().toString());
33         }
34         catch (InterruptedException ex)
35         {
36            resultJLabel.setText("Interrupted while waiting for results.");
37         }
38         catch (ExecutionException ex)
39         {
40            resultJLabel.setText(
41               "Error encountered while performing calculation.");
42         }
43      }
44
45      // recursive method fibonacci; calculates nth Fibonacci number
46      public long fibonacci(long number)
47      {
48         if (number == 0 || number == 1)
49            return number;
50         else
51            return fibonacci(number - 1) + fibonacci(number - 2);
52      }
53   } // end class BackgroundCalculator
```

Fig. 23.24 | `SwingWorker` subclass for calculating Fibonacci numbers in a background thread. (Part 2 of 2.)

`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 23.11.2.

A `BackgroundCalculator` object can be instantiated 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.

Software Engineering Observation 23.8

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. 23.25) 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 `JPanel`s. Lines 46–47 and 78–79 add two `JLabel`s, a `JTextField` and a `JButton` to the `workerJPanel` to allow the user to enter an integer whose Fibonacci number will be calculated by the `BackgroundWorker`. Lines 84–85 and 103 add two `JLabel`s and a `JButton` to the `eventThreadJPanel` 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 `JLabel`s display `count` and `n2` initially, so that the user will see the text `Fibonacci of 1: 1` in the `eventThreadJPanel` when the GUI starts.

```
1  // Fig. 23.25: FibonacciNumbers.java
2  // Using SwingWorker to perform a long calculation with
3  // results displayed in a GUI.
4  import java.awt.GridLayout;
5  import java.awt.event.ActionEvent;
6  import java.awt.event.ActionListener;
7  import javax.swing.JButton;
8  import javax.swing.JFrame;
9  import javax.swing.JPanel;
10 import javax.swing.JLabel;
```

Fig. 23.25 | Using `SwingWorker` to perform a long calculation with results displayed in a GUI. (Part 1 of 4.)

```
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");
      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());
               }
```

Fig. 23.25 | Using SwingWorker to perform a long calculation with results displayed in a GUI. (Part 2 of 4.)

```
               catch(NumberFormatException ex)
               {
                  // display an error message if the user did not
                  // enter an integer
                  fibonacciJLabel.setText("Enter an integer.");
                  return;
               }

               // 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 anonymous inner class
      ); // end call to addActionListener
      workerJPanel.add(goJButton);
      workerJPanel.add(fibonacciJLabel);

      // 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 anonymous inner class
      ); // end call to addActionListener
      eventThreadJPanel.add(nextNumberJButton);

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

Fig. 23.25 | Using `SwingWorker` to perform a long calculation with results displayed in a GUI. (Part 3 of 4.)

```
110
111    // main method begins program execution
112    public static void main(String[] args)
113    {
114       FibonacciNumbers application = new FibonacciNumbers();
115       application.setDefaultCloseOperation(EXIT_ON_CLOSE);
116    }
117 } // end class FibonacciNumbers
```

a) Begin calculating Fibonacci of 40 in the background

b) Calculating other Fibonacci values while Fibonacci of 40 continues calculating

c) Fibonacci of 40 calculation finishes

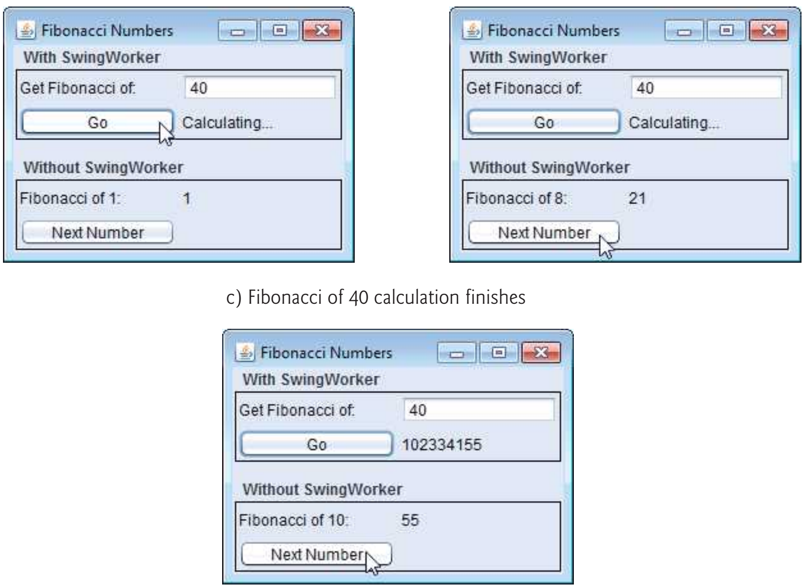


Fig. 23.25 | Using `SwingWorker` to perform a long calculation with results displayed in a GUI. (Part 4 of 4.)

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.

23.11.2 Processing Intermediate Results: Sieve of Eratosthenes

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 23.26 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. 23.26: 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.security.SecureRandom;
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.CancellationException;
import java.util.concurrent.ExecutionException;

public class PrimeCalculator extends SwingWorker<Integer, Integer>
{
   private static final SecureRandom generator = new SecureRandom();
   private final JTextArea intermediateJTextArea; // displays found primes
   private final JButton getPrimesJButton;
   private final JButton cancelJButton;
   private final JLabel statusJLabel; // displays status of calculation
   private final boolean[] primes; // boolean array for finding primes

   // constructor
   public PrimeCalculator(int max, JTextArea intermediateJTextArea,
      JLabel statusJLabel, JButton getPrimesJButton,
      JButton cancelJButton)
   {
      this.intermediateJTextArea = intermediateJTextArea;
      this.statusJLabel = statusJLabel;
      this.getPrimesJButton = getPrimesJButton;
      this.cancelJButton = cancelJButton;
      primes = new boolean[max];

```

Fig. 23.26 | Calculates the first *n* primes, displaying them as they are found. (Part 1 of 3.)

```
      Arrays.fill(primes, true); // initialize all primes elements to true
   }

   // finds all primes up to max using the Sieve of Eratosthenes
   public Integer doInBackground()
   {
      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));
            }
            catch (InterruptedException ex)
            {
               statusJLabel.setText("Worker thread interrupted");
               return count;
            }

            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
            }
         }
      }

      return count;
   }

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

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

Fig. 23.26 | Calculates the first *n* primes, displaying them as they are found. (Part 2 of 3.)

```
86        cancelJButton.setEnabled(false); // disable Cancel button
87
88        try
89        {
90           // retrieve and display doInBackground return value
91           statusJLabel.setText("Found " + get() + " primes.");
92        }
93        catch (InterruptedException | ExecutionException |
94           CancellationException ex)
95        {
96           statusJLabel.setText(ex.getMessage());
97        }
98     }
99  } // end class PrimeCalculator
```

Fig. 23.26 | Calculates the first *n* primes, displaying them as they are found. (Part 3 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 53 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 83–98 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 91 gets and displays the return value—the number of primes found—from method `doInBackground`. Lines 93–97 catch the exceptions thrown by method `get` and display an appropriate message in the `statusJLabel`.

Class `FindPrimes`

Class `FindPrimes` (Fig. 23.27) 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` shows the calculation's progress. The constructor (lines 32–125) sets up the GUI.

```
 1  // Fig. 23.27: FindPrimes.java
 2  // Using a SwingWorker to display prime numbers and update a JProgressBar
 3  // while the prime numbers are being calculated.
 4  import javax.swing.JFrame;
 5  import javax.swing.JTextField;
 6  import javax.swing.JTextArea;
 7  import javax.swing.JButton;
 8  import javax.swing.JProgressBar;
 9  import javax.swing.JLabel;
10  import javax.swing.JPanel;
11  import javax.swing.JScrollPane;
12  import javax.swing.ScrollPaneConstants;
```

Fig. 23.27 | Using a `SwingWorker` to display prime numbers and update a `JProgressBar` while the prime numbers are being calculated. (Part 1 of 4.)

```
import java.awt.BorderLayout;
import java.awt.GridLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.util.concurrent.ExecutionException;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeEvent;

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;

   // constructor
   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

               try
               {
                  // get user input
                  number = Integer.parseInt(
                     highestPrimeJTextField.getText());
               }
               catch (NumberFormatException ex)
               {
                  statusJLabel.setText("Enter an integer.");
                  return;
               }

```

Fig. 23.27 | Using a `SwingWorker` to display prime numbers and update a `JProgressBar` while the prime numbers are being calculated. (Part 2 of 4.)

```java
               // 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 anonymous inner class
               ); // end call to addPropertyChangeListener

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

               calculator.execute(); // execute the PrimeCalculator object
            }
         } // end anonymous inner class
      ); // end call to addActionListener
      northJPanel.add(getPrimesJButton);

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

      // initialize a panel to display cancelJButton,
      // progressJProgressBar, and statusJLabel
      JPanel southJPanel = new JPanel(new GridLayout(1, 3, 10, 10));
      cancelJButton.setEnabled(false);
      cancelJButton.addActionListener(
         new ActionListener()
         {
            public void actionPerformed(ActionEvent e)
            {
               calculator.cancel(true); // cancel the calculation
            }
         } // end anonymous inner class
      ); // end call to addActionListener
```

Fig. 23.27 | Using a SwingWorker to display prime numbers and update a JProgressBar while the prime numbers are being calculated. (Part 3 of 4.)

```
116         southJPanel.add(cancelJButton);
117         progressJProgressBar.setStringPainted(true);
118         southJPanel.add(progressJProgressBar);
119         southJPanel.add(statusJLabel);
120
121         add(northJPanel, BorderLayout.NORTH);
122         add(southJPanel, BorderLayout.SOUTH);
123         setSize(350, 300);
124         setVisible(true);
125      } // end constructor
126
127      // main method begins program execution
128      public static void main(String[] args)
129      {
130         FindPrimes application = new FindPrimes();
131         application.setDefaultCloseOperation(EXIT_ON_CLOSE);
132      } // end main
133   } // end class FindPrimes
```

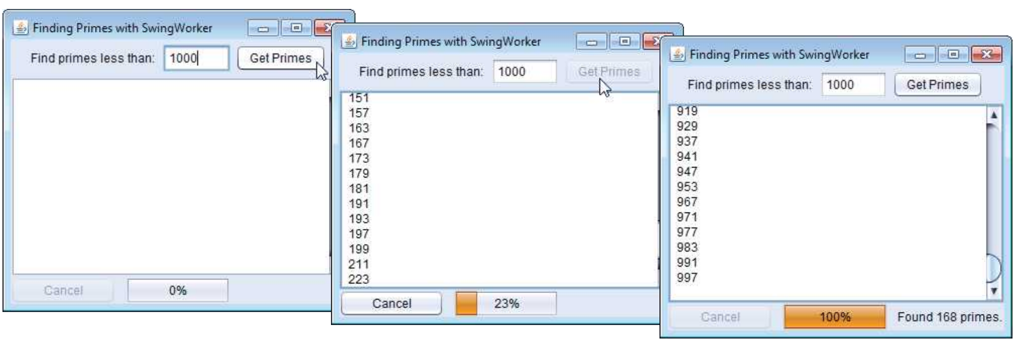


Fig. 23.27 | Using a `SwingWorker` to display prime numbers and update a `JProgressBar` while the prime numbers are being calculated. (Part 4 of 4.)

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`.

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`, 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 `PrimeCalculator` to begin finding primes. If the user clicks the `cancelJButton`, 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.

23.12 sort/parallelSort Timings with the Java SE 8 Date/Time API

In Section 7.15, we used class `Arrays`'s `static` method `sort` to sort an array and we introduced `static` method `parallelSort` for sorting large arrays more efficiently on multi-core systems. Figure 23.28 uses both methods to sort 15,000,000 element arrays of random `int` values so that we can demonstrate `parallelSort`'s performance improvement of over `sort` on a multi-core system (we ran this on a dual-core system).

```java
// SortComparison.java
// Comparing performance of Arrays methods sort and parallelSort.
import java.time.Duration;
import java.time.Instant;
import java.text.NumberFormat;
import java.util.Arrays;
import java.security.SecureRandom;

public class SortComparison
{
   public static void main(String[] args)
   {
      SecureRandom random = new SecureRandom();

      // create array of random ints, then copy it
      int[] array1 = random.ints(15_000_000).toArray();
      int[] array2 = new int[array1.length];
      System.arraycopy(array1, 0, array2, 0, array1.length);

      // time the sorting of array1 with Arrays method sort
      System.out.println("Starting sort");
      Instant sortStart = Instant.now();
      Arrays.sort(array1);
      Instant sortEnd = Instant.now();

      // display timing results
      long sortTime = Duration.between(sortStart, sortEnd).toMillis();
      System.out.printf("Total time in milliseconds: %d%n%n", sortTime);

      // time the sorting of array2 with Arrays method parallelSort
      System.out.println("Starting parallelSort");
      Instant parallelSortStart = Instant.now();
```

Fig. 23.28 | Comparing performance of `Arrays` methods `sort` and `parallelSort`. (Part 1 of 2.)

```
33         Arrays.parallelSort(array2);
34         Instant parallelSortEnd = Instant.now();
35
36         // display timing results
37         long parallelSortTime =
38            Duration.between(parallelSortStart, parallelSortEnd).toMillis();
39         System.out.printf("Total time in milliseconds: %d%n%n",
40            parallelSortTime);
41
42         // display time difference as a percentage
43         String percentage = NumberFormat.getPercentInstance().format(
44            (double) sortTime / parallelSortTime);
45         System.out.printf("%nsort took %s more time than parallelSort%n",
46            percentage);
47      }
48   } // end class SortComparison
```

```
Starting sort
Total time in milliseconds: 1319

Starting parallelSort
Total time in milliseconds: 323

sort took 408% more time than parallelSort
```

Fig. 23.28 | Comparing performance of `Arrays` methods `sort` and `parallelSort`. (Part 2 of 2.)

Creating the Arrays

Line 16 uses `SecureRandom` method `ints` to create an `IntStream` of 15,000,000 random `int` values, then calls `IntStream` method `toArray` to place the values into an array. Lines 17 and 18 copy the array so that the calls to both `sort` and `parallelSort` work with the same set of values.

Timing `Arrays` Method `sort` with Date/Time API Classes `Instant` and `Duration`

Lines 22 and 24 each call class `Instant`'s `static` method **`now`** to get the current time before and after the call to `sort`. To determine the difference between two `Instant`s, line 27 uses class `Duration`'s `static` method **`between`**, which returns a `Duration` object containing the time difference. Next, we call `Duration` method **`toMillis`** to get the difference in milliseconds.

Timing `Arrays` Method `parallelSort` with Date/Time API Classes `Instant` and `Duration`

Lines 32–34 time the call to `Arrays` method `parallelSort`. Then, lines 37–38 calculate the difference between the `Instant`s.

Displaying the Percentage Difference Between the Sorting Times

Lines 43–44 use a `NumberFormat` (package `java.text`) to format the ratio of the sort times as a percentage. `NumberFormat` `static` method **`getPercentInstance`** returns a `NumberFormat` that's used to format a number as a percentage. `NumberFormat` method `format` performs the formatting. As you can see in the sample output, the sort method took over *400% more time* to sort the 15,000,000 random `int` values.

Other Parallel Array Operations

In addition to method `parallelSort`, class `Arrays` now contains methods `parallelSetAll` and `parallelPrefix`, which perform the following tasks:

- **`parallelSetAll`**—Fills an array with values produced by a generator function that receives an `int` and returns a value of type `int`, `long` or `double`. Depending on which overload of method `parallelSetAll` is used, the generator function is an object of a class that implements `IntToDoubleFunction` (for `double` arrays), `IntUnaryOperator` (for `int` arrays), `IntToLongFunction` (for `long` arrays) or `IntFunction` (for arrays of any non-primitive type).
- **`parallelPrefix`**—Applies a `BinaryOperator` to the current and previous array elements and stores the result in the current element. For example, consider:

```
int[] values = {1, 2, 3, 4, 5};
Arrays.parallelPrefix(values, (x, y) -> x + y);
```

This call to `parallelPrefix` uses a `BinaryOperator` that *adds* two values. After the call completes, the array contains 1, 3, 6, 10 and 15. Similarly, the following call to `parallelPrefix`, uses a `BinaryOperator` that *multiplies* two values. After the call completes, the array contains 1, 2, 6, 24 and 120:

```
int[] values = {1, 2, 3, 4, 5};
Arrays.parallelPrefix(values, (x, y) -> x * y);
```

23.13 Java SE 8: Sequential vs. Parallel Streams

In Chapter 17, you learned about Java SE 8 lambdas and streams. We mentioned that streams are easy to *parallelize*, enabling programs to benefit from enhanced performance on multi-core systems. Using the timing capabilities introduced in Section 23.12, Fig. 23.29 demonstrates both *sequential* and *parallel* stream operations on a 10,000,000-element array of random `long` values (created at line 17) to compare the performance.

```
1  // StreamStatisticsComparison.java
2  // Comparing performance of sequential and parallel stream operations.
3  import java.time.Duration;
4  import java.time.Instant;
5  import java.util.Arrays;
6  import java.util.LongSummaryStatistics;
7  import java.util.stream.LongStream;
8  import java.security.SecureRandom;
9
10 public class StreamStatisticsComparison
11 {
12    public static void main(String[] args)
13    {
14       SecureRandom random = new SecureRandom();
15
16       // create array of random long values
17       long[] values = random.longs(10_000_000, 1, 1001).toArray();
```

Fig. 23.29 | Comparing performance of sequential and parallel stream operations. (Part 1 of 3.)

```java

      // perform calculcations separately
      Instant separateStart = Instant.now();
      long count = Arrays.stream(values).count();
      long sum = Arrays.stream(values).sum();
      long min = Arrays.stream(values).min().getAsLong();
      long max = Arrays.stream(values).max().getAsLong();
      double average = Arrays.stream(values).average().getAsDouble();
      Instant separateEnd = Instant.now();

      // display results
      System.out.println("Calculations performed separately");
      System.out.printf("    count: %,d%n", count);
      System.out.printf("      sum: %,d%n", sum);
      System.out.printf("      min: %,d%n", min);
      System.out.printf("      max: %,d%n", max);
      System.out.printf("  average: %f%n", average);
      System.out.printf("Total time in milliseconds: %d%n%n",
         Duration.between(separateStart, separateEnd).toMillis());

      // time sum operation with sequential stream
      LongStream stream1 = Arrays.stream(values);
      System.out.println("Calculating statistics on sequential stream");
      Instant sequentialStart = Instant.now();
      LongSummaryStatistics results1 = stream1.summaryStatistics();
      Instant sequentialEnd = Instant.now();

      // display results
      displayStatistics(results1);
      System.out.printf("Total time in milliseconds: %d%n%n",
         Duration.between(sequentialStart, sequentialEnd).toMillis());

      // time sum operation with parallel stream
      LongStream stream2 = Arrays.stream(values).parallel();
      System.out.println("Calculating statistics on parallel stream");
      Instant parallelStart = Instant.now();
      LongSummaryStatistics results2 = stream2.summaryStatistics();
      Instant parallelEnd = Instant.now();

      // display results
      displayStatistics(results1);
      System.out.printf("Total time in milliseconds: %d%n%n",
         Duration.between(parallelStart, parallelEnd).toMillis());
   }

   // display's LongSummaryStatistics values
   private static void displayStatistics(LongSummaryStatistics stats)
   {
      System.out.println("Statistics");
      System.out.printf("    count: %,d%n", stats.getCount());
      System.out.printf("      sum: %,d%n", stats.getSum());
      System.out.printf("      min: %,d%n", stats.getMin());
      System.out.printf("      max: %,d%n", stats.getMax());
```

Fig. 23.29 | Comparing performance of sequential and parallel stream operations. (Part 2 of 3.)

```
71          System.out.printf("  average: %f%n", stats.getAverage());
72       }
73    } // end class StreamStatisticsComparison
```

```
Calculations performed separately
    count: 10,000,000
      sum: 5,003,695,285
      min: 1
      max: 1,000
  average: 500.369529
Total time in milliseconds: 173

Calculating statistics on sequential stream
Statistics
    count: 10,000,000
      sum: 5,003,695,285
      min: 1
      max: 1,000
  average: 500.369529
Total time in milliseconds: 69

Calculating statistics on parallel stream
Statistics
    count: 10,000,000
      sum: 5,003,695,285
      min: 1
      max: 1,000
  average: 500.369529
Total time in milliseconds: 38
```

Fig. 23.29 | Comparing performance of sequential and parallel stream operations. (Part 3 of 3.)

Performing Stream Operations with Separate Passes of a Sequential Stream

Section 17.3 demonstrated various numerical operations on `IntStream`s. Lines 20–26 perform and time the `count`, `sum`, `min`, `max` and `average` stream operations each performed individually on a `LongStream` returned by `Arrays` method `stream`. Lines 29–36 then display the results and the total time required to perform all five operations.

Performing Stream Operations with a Single Pass of a Sequential Stream

Lines 39–48 demonstrate the performance improvement you get by using `LongStream` method `summaryStatistics` to determine the count, sum, minimum value, maximum value and average in one pass of a *sequential* `LongStream`—all streams are sequential by default. This operation took approximately 40% of the time required to perform the five operations separately.

Performing Stream Operations with a Single Pass of a Parallel Stream

Lines 51–60 demonstrate the performance improvement you get by using `LongStream` method `summaryStatistics` on a *parallel* `LongStream`. To obtain a parallel stream that can take advantage of multi-core processors, simply invoke method `parallel` on an existing stream. As you can see from the sample output, performing the operations on a parallel stream decreased the total time required even further—taking approximately 55% of the calculation time for the sequential `LongStream` and just 22% of the time required to perform the five operations separately.

23.14 (Advanced) Interfaces `Callable` and `Future`

Interface `Runnable` provides only the most basic functionality for multithreaded programming. In fact, this interface has limitations. 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 *shared mutable data* would be required to pass the value back to the calling thread. As you now know, this would require thread synchronization. The **`Callable`** interface (of package `java.util.concurrent`) fixes this limitation. The interface declares a single method named **`call`** which returns a value representing the result of the `Callable`'s task—such as the result of a long running calculation.

An application that creates a `Callable` likely wants to run it concurrently with other `Runnable`s and `Callable`s. `ExecutorService` method **`submit`** executes its `Callable` argument and returns an object of type **`Future`** (of package `java.util.concurrent`), which represents the `Callable`'s future result. The `Future` interface **`get`** method *blocks* the calling thread, and waits for the `Callable` to complete and return its result. The interface also provides methods that enable you to cancel a `Callable`'s execution, determine whether the `Callable` was cancelled and determine whether the `Callable` completed its task.

Executing Aysnchronous Tasks with `CompletableFuture`

Java SE 8 introduces class **`CompletableFuture`** (package `java.util.concurrent`), which implements the `Future` interface and enables you to *asynchronously* execute `Runnable`s that perform tasks or `Supplier`s that return values. Interface **`Supplier`**, like interface `Callable`, is a functional interface with a single method (in this case, `get`) that receives no arguments and returns a result. Class `CompletableFuture` provides many additional capabilities that for advanced programmers, such as creating `CompletableFuture`s without executing them immediately, composing one or more `CompletableFuture`s so that you can wait for any or all of them to complete, executing code after a `CompletableFuture` completes and more.

Figure 23.30 performs two long-running calculations sequentially, then performs them again asynchronously using `CompletableFuture`s to demonstrate the performance improvement from asynchronous execution on a multi-core system. For demonstration purposes, our long-running calculation is performed by a recursive `fibonacci` method (lines 73–79; similar to the one presented in Section 18.5). For larger Fibonacci values, the recursive implementation can require *significant* computation time—in practice, it's much faster to calculate Fibonacci values using a loop.

```
1  // FibonacciDemo.java
2  // Fibonacci calculations performed synchronously and asynchronously
3  import java.time.Duration;
4  import java.text.NumberFormat;
5  import java.time.Instant;
6  import java.util.concurrent.CompletableFuture;
7  import java.util.concurrent.ExecutionException;
8
```

Fig. 23.30 | Fibonacci calculations performed synchronously and asynchronously. (Part 1 of 4.)

```java
// class that stores two Instants in time
class TimeData
{
   public Instant start;
   public Instant end;

   // return total time in seconds
   public double timeInSeconds()
   {
      return Duration.between(start, end).toMillis() / 1000.0;
   }
} // end class TimeData

public class FibonacciDemo
{
   public static void main(String[] args)
      throws InterruptedException, ExecutionException
   {
      // perform synchronous fibonacci(45) and fibonacci(44) calculations
      System.out.println("Synchronous Long Running Calculations");
      TimeData synchronousResult1 = startFibonacci(45);
      TimeData synchronousResult2 = startFibonacci(44);
      double synchronousTime =
         calculateTime(synchronousResult1, synchronousResult2);
      System.out.printf(
         "  Total calculation time = %.3f seconds%n", synchronousTime);

      // perform asynchronous fibonacci(45) and fibonacci(44) calculations
      System.out.printf("%nAsynchronous Long Running Calculations%n");
      CompletableFuture<TimeData> futureResult1 =
         CompletableFuture.supplyAsync(() -> startFibonacci(45));
      CompletableFuture<TimeData> futureResult2 =
         CompletableFuture.supplyAsync(() -> startFibonacci(44));

      // wait for results from the asynchronous operations
      TimeData asynchronousResult1 = futureResult1.get();
      TimeData asynchronousResult2 = futureResult2.get();
      double asynchronousTime =
         calculateTime(asynchronousResult1, asynchronousResult2);
      System.out.printf(
         "  Total calculation time = %.3f seconds%n", asynchronousTime);

      // display time difference as a percentage
      String percentage = NumberFormat.getPercentInstance().format(
         synchronousTime / asynchronousTime);
      System.out.printf("%nSynchronous calculations took %s" +
         " more time than the asynchronous calculations%n", percentage);
   }

   // executes function fibonacci asynchronously
   private static TimeData startFibonacci(int n)
   {
```

Fig. 23.30 | Fibonacci calculations performed synchronously and asynchronously. (Part 2 of 4.)

```
      // create a TimeData object to store times
      TimeData timeData = new TimeData();

      System.out.printf("  Calculating fibonacci(%d)%n", n);
      timeData.start = Instant.now();
      long fibonacciValue = fibonacci(n);
      timeData.end = Instant.now();
      displayResult(n, fibonacciValue, timeData);
      return timeData;
   }

   // recursive method fibonacci; calculates nth Fibonacci number
   private static long fibonacci(long n)
   {
      if (n == 0 || n == 1)
         return n;
      else
         return fibonacci(n - 1) + fibonacci(n - 2);
   }

   // display fibonacci calculation result and total calculation time
   private static void displayResult(int n, long value, TimeData timeData)
   {
      System.out.printf("  fibonacci(%d) = %d%n", n, value);
      System.out.printf(
         "  Calculation time for fibonacci(%d) = %.3f seconds%n",
         n, timeData.timeInSeconds());
   }

   // display fibonacci calculation result and total calculation time
   private static double calculateTime(TimeData result1, TimeData result2)
   {
      TimeData bothThreads = new TimeData();

      // determine earlier start time
      bothThreads.start = result1.start.compareTo(result2.start) < 0 ?
         result1.start : result2.start;

      // determine later end time
      bothThreads.end = result1.end.compareTo(result2.end) > 0 ?
         result1.end : result2.end;

      return bothThreads.timeInSeconds();
   }
} // end class FibonacciDemo
```

```
Synchronous Long Running Calculations
  Calculating fibonacci(45)
  fibonacci(45) = 1134903170
  Calculation time for fibonacci(45) = 5.884 seconds
  Calculating fibonacci(44)
  fibonacci(44) = 701408733
```

Fig. 23.30 | Fibonacci calculations performed synchronously and asynchronously. (Part 3 of 4.)

```
  Calculation time for fibonacci(44) = 3.605 seconds
  Total calculation time = 9.506 seconds

Asynchronous Long Running Calculations
  Calculating fibonacci(45)
  Calculating fibonacci(44)
  fibonacci(44) = 701408733
  Calculation time for fibonacci(44) = 3.650 seconds
  fibonacci(45) = 1134903170
  Calculation time for fibonacci(45) = 5.911 seconds
  Total calculation time = 5.911 seconds

Synchronous calculations took 161% more time than the asynchronous ones
```

Fig. 23.30 | Fibonacci calculations performed synchronously and asynchronously. (Part 4 of 4.)

Class `TimeData`

Class `TimeData` (lines 10–20) stores two `Instant`s representing the start and end time of a task, and provides method `timeInSeconds` to calculate the total time between them. We use `TimeData` objects throughout this example to calculate the time required to perform Fibonacci calculations.

Method `startFibonacci` for Performing and Timing Fibonacci Calculations

Method `startFibonacci` (lines 59–70) is called several times in `main` (lines 29, 30, 39 and 41) to initiate Fibonacci calculations and to calculate the time each calculation requires. The method receives the Fibonacci number to calculate and performs the following tasks:

- Line 62 creates a `TimeData` object to store the calculation's start and end times.
- Line 64 displays the Fibonacci number to be calculated.
- Line 65 stores the current time before method `fibonacci` is called.
- Line 66 calls method `fibonacci` to perform the calculation.
- Line 67 stores the current time after the call to `fibonacci` completes.
- Line 68 displays the result and the total time required for the calculation.
- Line 69 returns the `TimeData` object for use in method `main`.

Performing Fibonacci Calculations Synchronously

Method `main` (lines 24–56) first demonstrates synchronous Fibonacci calculations. Line 29 calls `startFibonacci(45)` to initiate the `fibonacci(45)` calculation and store the `TimeData` object containing the calculation's start and end times. When this call completes, line 30 calls `startFibonacci(44)` to initiate the `fibonacci(44)` calculation and store its `TimeData`. Next, lines 31–32 pass both `TimeData` objects to method `calculateTime` (lines 91–104), which returns the total calculation time in seconds. Lines 33–34 display the total calculation time for the synchronous Fibonacci calculations.

Performing Fibonacci Calculations Asynchronously

Lines 38–41 in `main` launch the asynchronous Fibonacci calculations in separate threads. `CompletableFuture` `static` method **`supplyAsync`** executes an asynchronous task that returns a value. The method receives as its argument an object that implements interface `Sup-`

`plier`—in this case, we use a lambdas with empty parameter lists to invoke `startFibonacci(45)` (line 39) and `startFibonacci(44)` (line 41). The compiler infers that `supplyAsync` returns a `CompletableFuture<TimeData>` because method `startFibonacci` returns type `TimeData`. Class `CompletableFuture` also provides `static` method **`runAsync`** to execute an asynchronous task that does not return a result—this method receives a `Runnable`.

Getting the Asynchronous Calculations' Results

Class `CompletableFuture` implements interface `Future`, so we can obtain the asynchronous tasks' results by calling `Future` method `get` (lines 44–45). These are *blocking* calls—they cause the `main` thread to *wait* until the asynchronous tasks complete and return their results. In our case, the results are `TimeData` objects. Once both tasks return, lines 46–47 pass both `TimeData` objects to method `calculateTime` (lines 91–104) to get the total calculation time in seconds. Then, lines 48–49 display the total calculation time for the asynchronous Fibonacci calculations. Finally, lines 52–55 calculate and display the percentage difference in execution time for the synchronous and asynchronous calculations.

Program Outputs

On our dual-core computer, the synchronous calculations took a total of 9.506 seconds. Though the individual asynchronous calculations took approximately the same amount of time as the corresponding synchronous calculations, the total time for the asynchronous calculations was only 5.911 seconds, because the two calculations were actually performed *in parallel*. As you can see in the output, the synchronous calculations took 161% more time to complete, so asynchronous execution provided a significant performance improvement.

23.15 (Advanced) Fork/Join Framework

Java's concurrency APIs include the 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 nevertheless benefit by the fork/join framework's use "behind the scenes" in the Java API and other third party libraries. For example, the parallel capabilities of Java SE 8 streams are implemented using this framework.

The fork/join framework is particularly well suited to divide-and-conquer-style algorithms, such as the merge sort that we implemented in Section 19.8. Recall that the recursive merge-sort 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 framework can be used to create concurrent tasks so that they can be distributed across multiple processors and be truly performed in parallel—the details of assigning the tasks to different processors are handled for you by the framework.

23.16 Wrap-Up

In this chapter, we presented Java's concurrency capabilities for enhancing application performance on multi-core systems. You learned the differences between concurrent and parallel execution. We discussed that Java makes concurrency available to you through multithreading. 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 mutable 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.

We revisited the `Arrays` class's `sort` and `parallelSort` methods to demonstrate the benefit of using a parallel sorting algorithm on a multi-core processor. We used the Java SE 8 Date/Time API's Instant and Duration classes to time the sort operations.

You learned that Java SE 8 streams are easy to parallelize, enabling programs to benefit from enhanced performance on multi-core systems, and that to obtain a parallel stream, you simply invoke method `parallel` on an existing stream.

We discussed the `Callable` and `Future` interfaces, which enable you to execute tasks that return results and to obtain those results, respectively. We then presented an example of performing long-running tasks synchronously and asynchronously using Java SE 8's new `CompletableFuture` class. We use the multithreading techniques introduced in this chapter again in Chapter 28, Networking, to help build multithreaded servers that can interact with multiple clients concurrently. In the next chapter, we introduce database-application development with Java's JDBC API.

Summary

Section 23.1 Introduction

- Two tasks that are operating concurrently are both making progress at once.
- Two tasks that are operating in parallel are executing simultaneously. In this sense, parallelism is a subset of concurrency. Today's multi-core computers have multiple processors that can perform tasks in parallel.
- 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. This capability is called multithreading.
- In a multithreaded application, threads can be distributed across multiple processors (if available) so that multiple tasks execute in parallel and the application can operate more efficiently.
- The JVM creates threads to run a program and for housekeeping tasks such as garbage collection.
- Multithreading can also increase performance on single-processor systems—when one thread cannot proceed (because, for example, it's waiting for the result of an I/O operation), another can use the processor.
- The vast majority of programmers should use existing collection classes and interfaces from the concurrency APIs that manage synchronization for you.

Section 23.2 Thread States and Life Cycle

- A new thread begins its life cycle in the *new* state (p. 961). 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. 961) 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. 961) 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. 961) 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. 961) 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. 961) when it successfully completes its task or otherwise terminates (perhaps due to an error).
- At the operating-system level, the *runnable* state (p. 962) encompasses two separate states. When a thread first transitions to the *runnable* state from the *new* state, it's in the *ready* state (p. 962). A *ready* thread enters the *running* state (p. 962) when the operating system dispatches it.
- Most operating systems allot a quantum (p. 962) 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. 962) 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. 963).
- Depending on the operating system, higher-priority threads could postpone—possibly indefinitely (p. 963)—the execution of lower-priority threads.

Section 23.3 Creating and Executing Threads with the `Executor` Framework

- A `Runnable` (p. 963) object represents a task that can execute concurrently with other tasks.
- Interface `Runnable` declares method `run` (p. 963) 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. 963) to manage the execution of `Runnable` objects. An `Executor` object typically creates and manages a group of threads—called a thread pool (p. 963).
- `Executors` (p. 964) can reuse existing threads and can improve performance by optimizing the number of threads to ensure that the processor stays busy.
- `Executor` method `execute` (p. 963) 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. 964) 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. 965) 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. 967) notifies the ExecutorService to stop accepting new tasks, but continues executing existing tasks and terminates when those tasks complete execution.

Section 23.4 Thread Synchronization

- Thread synchronization (p. 968) coordinates access to shared mutable 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. 968).
- A common way to perform synchronization is to use Java's built-in monitors. Every object has a monitor and a monitor lock (p. 968). 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. 968) 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. 968). Such code is said to be guarded by the monitor lock.
- 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.
- Java also allows `synchronized` methods (p. 969). Before executing, a `synchronized` instance method must acquire the lock on the object that's used to call the method. Similary, a `static synchronized` method must acquire the lock on the class that's used to call the method.
- `ExecutorService` method `awaitTermination` (p. 973) 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. 974) 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 23.5 Producer/Consumer Relationship without Synchronization

- In a multithreaded producer/consumer relationship (p. 976), 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 23.6 Producer/Consumer Relationship: `ArrayBlockingQueue`

- `ArrayBlockingQueue` (p. 984) 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. 984) places an element at the end of the `BlockingQueue`, waiting if the queue is full. Method `take` (p. 984) removes an element from the head of the `BlockingQueue`, waiting if the queue is empty.
- `ArrayBlockingQueue` stores shared mutable data in an array that's sized with an argument passed to the constructor. Once created, an `ArrayBlockingQueue` is fixed in size.

Section 23.7 (Advanced) Producer/Consumer Relationship with `synchronized`*,* `wait`*,* `notify` *and* `notifyAll`

- You can implement a shared buffer yourself using the `synchronized` keyword and `Object` methods `wait`, `notify` and `notifyAll` (p. 988).
- 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. 988) 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. 988), 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 23.8 (Advanced) Producer/Consumer Relationship: Bounded Buffers

- You cannot make assumptions about the relative speeds of concurrent threads.
- A bounded buffer (p. 995) 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 23.9 (Advanced) Producer/Consumer Relationship: The `Lock` *and* `Condition` *Interfaces*

- The `Lock` and `Condition` interfaces (p. 1002) 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 a `Lock`'s `lock` method (p. 1002) to acquire the lock. Once a `Lock` has been obtained by one thread, the `Lock` will not allow another thread to obtain it until the first thread releases it (by calling the `Lock`'s `unlock` method; p. 1002).
- 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. 1002) is a basic implementation of the `Lock` interface.
- The `ReentrantLock` constructor takes a `boolean` that specifies whether the lock has a fairness policy (p. 1002). 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. 1003). Using `Lock` objects allows you to explicitly declare the condition objects on which a thread may need to wait.
- `Condition` (p. 1003) 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 `signallAll` methods (p. 1003). With `synchronized`, there's no way to explicitly state the condition on which threads are waiting.

Section 23.11 Multithreading with GUI: `SwingWorker`

- The event dispatch thread (p. 1011) 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` (package `javax.swing`; p. 1011), which implements `Runnable`, to perform a task in a worker thread then update Swing components from the event dispatch thread based on the task's 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. 1024) 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 23.12 Timing `sort` and `parallelSort` with the Java SE 8 Date/Time API

- Class `Instant`'s `static` method `now` gets the current time.
- To determine the difference between two `Instant`s, use class `Duration`'s `static` method `between`, which returns a `Duration` object containing the time difference.
- `Duration` method `toMillis` returns the `Duration` as a `long` value milliseconds.
- `NumberFormat` `static` method `getPercentInstance` returns a `NumberFormat` that's used to format a number as a percentage.
- `NumberFormat` method `format` returns a `String` representation of its argument in the specified numeric format.
- `Arrays` `static` method `parallelSetAll` fills an array with values produced by a generator function that receives an `int` and returns a value of type `int`, `long` or `double`. Depending on which overload of method `parallelSetAll` is used the generator function is an object of a class that implements `IntToDoubleFunction` (for double arrays), `IntUnaryOperator` (for int arrays), `IntToLongFunction` (for long arrays) or `IntFunction` (for arrays of any non-primitive type).
- `Arrays` `static` method `parallelPrefix` applies a `BinaryOperator` to the current and previous array elements and stores the result in the current element.

Section 23.13 Java SE 8: Sequential vs. Parallel Streams

- Streams are easy to parallelize, enabling programs to benefit from enhanced performance on multi-core systems.
- To obtain a parallel stream, simply invoke method `parallel` on an existing stream.

Section 23.14 (Advanced) Interfaces `Callable` and `Future`

- The `Callable` (p. 1030) interface (of package `java.util.concurrent`) declares a single method named `call` that allows a task to return a value.
- `ExecutorService` method `submit` (p. 1030) executes a `Callable` passed in as its argument. Method `submit` returns an object of type `Future` (of package `java.util.concurrent`) that represents the future result of the executing `Callable`.
- Interface `Future` (p. 1030) declares method `get` to return the result of the `Callable`. The interface also provides methods that enable you to cancel a `Callable`'s execution, determine whether the `Callable` was cancelled and determine whether the `Callable` completed its task.
- Java SE 8 introduces a new `CompletableFuture` class (package `java.util.concurrent`; p. 1030), which implements the `Future` interface and enables you to asynchronously execute `Runnable`s that perform tasks or `Supplier`s that return values.
- Interface `Supplier` (p. 1030), like interface `Callable`, is a functional interface with a single method (in this case, `get`) that receives no arguments and returns a result.
- `CompletableFuture` `static` method `supplyAsync` (p. 1033) asynchronously executes a `Supplier` task that returns a value.
- `CompletableFuture` `static` method `runAsync` (p. 1034) asynchronously executes a `Runnable` task that does not return a result.
- `CompletableFuture` method `get` is a blocking method—it causes the calling thread to wait until the asynchronous task completes and returns its results.

Section 23.15 (Advanced) Fork/Join Framework

- Java's concurrency APIs include the 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

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

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

e) `Runnable`s are executed using a class that implements the ________ interface.

f) `ExecutorService` method ________ ends each thread in an `ExecutorService` as soon as it finishes executing its current `Runnable`, if any.

g) In a(n) ________ relationship, the ________ generates data and stores it in a shared object, and the ________ reads data from the shared object.

h) Keyword ________ indicates that only one thread at a time should execute on an object.

23.2 *(Advanced Optional Sections)* Fill in the blanks in each of the following statements:

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

b) Method ________ of class `Condition` moves every thread in an object's *waiting* state to the *runnable* state.

c) A thread can call method ________ on a `Condition` object to release the associated `Lock` and place that thread in the ________ state.

d) Class ________ implements the `BlockingQueue` interface using an array.
e) Class `Instant`'s `static` method ________ gets the current time.
f) `Duration` method ________ returns the `Duration` as a `long` value milliseconds.
g) `NumberFormat` `static` method ________ returns a `NumberFormat` that's used to format a number as a percentage.
h) `NumberFormat` method ________ returns a `String` representation of its argument in the specified numeric format.
i) `Arrays` `static` method ________ fills an array with values produced by a generator function.
j) `Arrays` `static` method ________ applies a `BinaryOperator` to the current and previous array elements and stores the result in the current element.
k) To obtain a parallel stream, simply invoke method ________ on an existing stream.
l) Among its many features a `CompletableFuture` enables you to asynchronously execute ________ that perform tasks or ________ that return values.

23.3 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.

23.4 *(Advanced Optional Sections)* State whether each of the following is *true* or *false*. If *false*, explain why.
a) To determine the difference between two `Instant`s, use class `Duration`'s `static` method `difference`, which returns a `Duration` object containing the time difference.
b) Streams are easy to parallelize, enabling programs to benefit from enhanced performance on multi-core systems.
c) Interface `Supplier`, like interface `Callable`, is a functional interface with a single method that receives no arguments and returns a result.
d) `CompletableFuture` `static` method `runAsync` asynchronously executes a `Supplier` task that returns a value.
e) `CompletableFuture` `static` method `supplyAsync` asynchronously executes a `Runnable` task that does not return a result.

Answers to Self-Review Exercises

23.1 a) its `run` method ends. b) `sleep`, `Thread`. c) *timed waiting*. d) *ready*, *running*. e) `Executor`. f) `shutdown`. g) producer/consumer, producer, consumer. h) `synchronized`.

23.2 a) `signal`. b) `signalAll`. c) `await`, *waiting*. d) `ArrayBlockingQueue`. e) `now`. f) `toMillis`. g) `getPercentInstance`. h) `format`. i) `parallelSetAll`. j) `parallelPrefix`. k) `parallel`. l) `Runnables`, `Suppliers`.

23.3 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.

23.4 a) False. The `Duration` method for calculating the difference between two `Instants` is named `between`. b) True. c) True. d) False. The method that asynchronously executes a `Supplier` is `supplyAsync`. e) False. The method that asynchronously executes a `Runnable` is `runAsync`.

Exercises

23.5 *(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) Swing components are thread safe.
c) *(Advanced)* Declaring a method `synchronized` guarantees that deadlock cannot occur.
d) *(Advanced)* 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.

23.6 *(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) producer/consumer relationship
h) quantum

23.7 *(Advanced: Multithreading Terms)* Discuss each of the following terms in the context of Java's threading mechanisms:

a) `synchronized`
b) `wait`
c) `notify`
d) `notifyAll`
e) `Lock`
f) `Condition`

23.8 *(Blocked State)* 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.

23.9 *(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.

23.10 *(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`.

23.11 *(Bouncing Balls)* Modify the program in Exercise 23.10 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.

23.12 *(Bouncing Balls with Shadows)* Modify the program in Exercise 23.11 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`.

23.13 *(Advanced: Circular Buffer with `Locks` and `Conditions`)* Reimplement the example in Section 23.8 using the `Lock` and `Condition` concepts presented in Section 23.9.

23.14 *(Bounded Buffer: A Real-World Example)* Describe how a highway off-ramp onto a local road is a good example of a producer/consumer relationship with a bounded buffer. In particular, discuss how the designers might choose the size of the off-ramp.

Parallel Streams

For Exercises 23.15–23.17, you may need to create larger data sets to see a significant performance difference.

23.15 ***(Summarizing the Words in a File)*** Reimplement Fig. 17.17 using parallel streams. Use the Date/Time API timing techniques you learned in Section 23.12 to compare the time required for the sequential and parallel versions of the program.

23.16 ***(Summarizing the Characters in a File)*** Reimplement Exercise 17.9 using parallel streams. Use the Date/Time API timing techniques you learned in Section 23.12 to compare the time required for the sequential and parallel versions of the program.

23.17 ***(Summarizing the File Types in a Directory)*** Reimplement Exercise 17.10 using parallel streams. Use the Date/Time API timing techniques you learned in Section 23.12 to compare the time required for the sequential and parallel versions of the program.