Chapter 23 Concurrency

Java How to Program, 11/e

Questions? E-mail paul.deitel@deitel.com

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.

OBJECTIVES

- Use JavaFX's concurrency APIs to update 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.

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

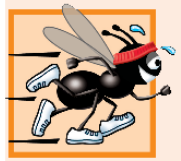
23.6 Producer/Consumer Relationship: `ArrayBlockingQueue`

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 in JavaFX
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

23.1 Introduction

- When we say that two tasks are operating concurrently, we mean that they're both *making progress* at once.
- When we say that two tasks are operating in parallel, we mean that they're executing *simultaneously*.
- Java makes concurrency available to you through the language and APIs.
- You specify that an application contains separate threads of execution
  - each thread has its own method-call stack and program counter
  - can execute concurrently with other threads while sharing application-wide resources such as memory and file handles.
- This capability is called multithreading.



Performance Tip 23.1

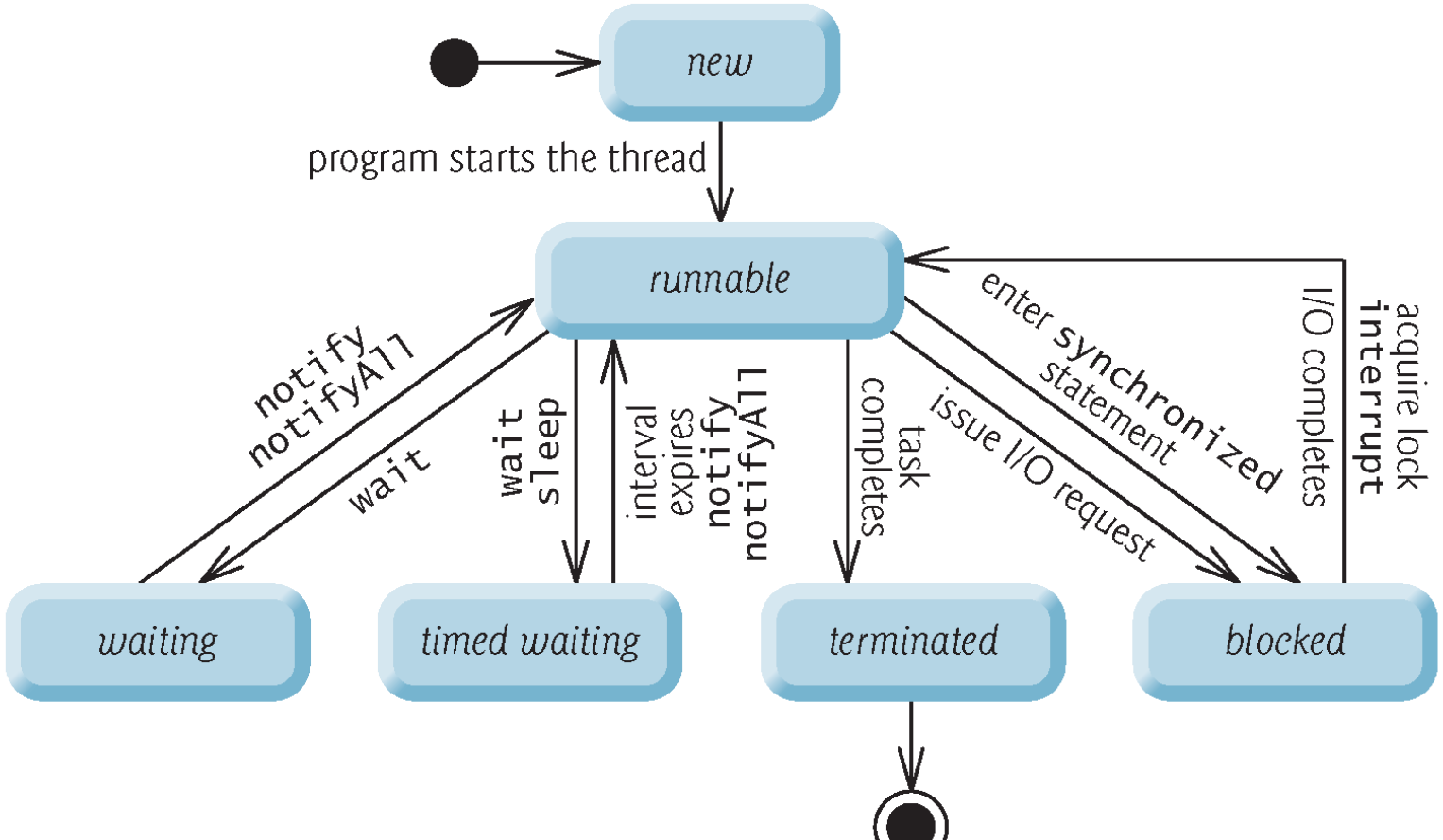
A problem with single-threaded applications that can lead to poor responsiveness is that lengthy activities must complete before others can begin. In a multi-threaded application, threads can be distributed across multiple cores (if available) so that multiple tasks execute in parallel and the application 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.

23.1 Introduction (cont.)

- Programming concurrent applications is difficult and error prone.
- Guidelines:
  - The vast majority of programmers should use existing collection classes and interfaces from the concurrency APIs that manage synchronization for you.
  - For advanced programmers who want to control synchronization, use the `synchronized` keyword and `Object` methods `wait`, `notify` and `notifyAll`.
  - Only the most advanced programmers should use `Locks` and `Conditions`

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.



23.2.1 *New* and *Runnable* States

- A new thread begins its life cycle in the new **state.**
- Remains there until started, which places it in the runnable **state—considered to be executing its task.**

23.2.2 *Waiting* State

- A *runnable* thread can transition to the *waiting* state while it waits for another thread to perform a task.
  - 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.
  - Transitions back to the *runnable* state when that time interval expires or when the event it's waiting for occurs.
  - Cannot use a processor, even if one is available.
- 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.

23.2.4 *Blocked* State

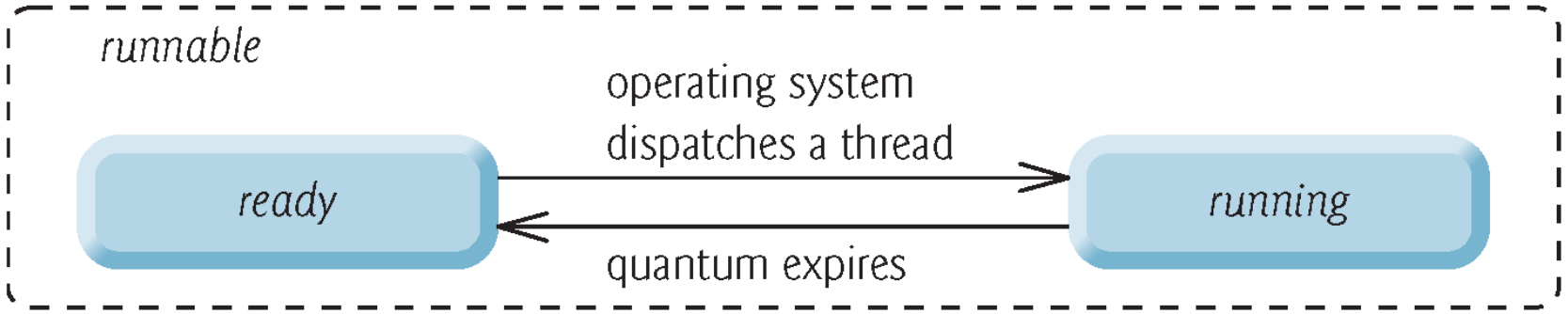
- A *runnable* thread transitions to the *blocked* state when it attempts to perform a task that cannot be completed immediately and it must temporarily wait until that task completes.

23.2.5 *Terminated* State

- A *runnable* thread enters the *terminated* state when it successfully completes its task or otherwise terminates (perhaps due to an error).

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).
- When a thread first transitions to the *runnable* state from the new state, it is 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.
- Typically, each thread is given a a quantum or timeslice in which to perform its task.
- The process that an operating system uses to determine which thread to dispatch is called thread scheduling.



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.
- Informally, higher-priority threads are more important to a program and should be allocated processor time before lower-priority threads.
- Thread priorities cannot guarantee the order in which threads execute.
- Do not explicitly create and use Thread objects to implement concurrency.
- Rather, use the `Executor` interface (described in Section 23.3).

23.2.7 Thread Priorities and Thread Scheduling (Cont.)

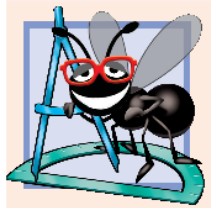
- Most operating systems support timeslicing, which enables threads of equal priority to share a processor.
- An operating system's thread scheduler determines which thread runs next.
- One simple thread-scheduler implementation keeps the highest-priority thread running at all times and, if there's more 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.

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).
- A steady influx of higher-priority threads could postpone—possibly indefinitely—the execution of lower-priority threads.
- Indefinite postponement is sometimes referred to as starvation.
- Operating systems employ a technique called *aging* to prevent starvation.

23.2.8 Indefinite Postponement and Deadlock (Cont.)

- 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.



Software Engineering Observation 23.1

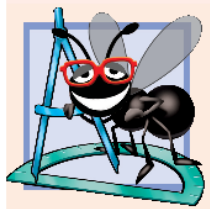
Java provides higher-level concurrency utilities to hide much of this complexity and make multithreaded programming less error prone. Thread priorities are used behind the scenes to interact with the operating system, but most programmers who use Java multithreading will not be concerned with setting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

23.3 Creating and Executing Threads with the Executor Framework

- A `Runnable` object represents 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.
- When a thread executing a `Runnable` is created and started, the thread calls the `Runnable` object's `run` method, which executes in the new thread.



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.

23.3 Creating and Executing Threads with the Executor Framework (cont.)

- Class `PrintTask` (Fig. 23.3) implements `Runnable`, so that multiple `PrintTasks` can execute concurrently.
- `Thread` `static` method `sleep` places a thread in the *timed waiting state for the specified amount of time.*
  - Can throw a checked exception of type `InterruptedException` if the sleeping thread's `interrupt` method is called.
- The code in `main` executes in the main thread, a thread created by the JVM.
- The code in the `run` method of `PrintTask` executes in the threads created in `main`.
- When method `main` terminates, the program itself continues running because there are still threads that are alive.
  - The program will not terminate until its last thread completes execution.

```
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     private static final SecureRandom generator = new SecureRandom();
7     private final int sleepTime; // random sleep time for thread
8     private final String taskName;
9
10    // constructor
11    public PrintTask(String taskName) {
12       this.taskName = taskName;
13
14       // pick random sleep time between 0 and 5 seconds
15       sleepTime = generator.nextInt(5000); // milliseconds
16    }
17
```

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

```
18      // method run contains the code that a thread will execute
19      @Override
20      public void run() {
21         try { // put thread to sleep for sleepTime amount of time
22            System.out.printf("%s going to sleep for %d milliseconds.%n",
23               taskName, sleepTime);
24            Thread.sleep(sleepTime); // put thread to sleep
25         }
26         catch (InterruptedException exception) {
27            exception.printStackTrace();
28            Thread.currentThread().interrupt(); // re-interrupt the thread
29         }
30
31         // print task name
32         System.out.printf("%s done sleeping%n", taskName);
33      }
34   }
```

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

```
1  // Fig. 23.4: TaskExecutor.java
2  // Using an ExecutorService to execute Runnables.
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.ExecutorService;
5
6  public class TaskExecutor {
7     public static void main(String[] args) {
8        // create and name each runnable
9        PrintTask task1 = new PrintTask("task1");
10       PrintTask task2 = new PrintTask("task2");
11       PrintTask task3 = new PrintTask("task3");
12
13       System.out.println("Starting Executor");
14
15       // create ExecutorService to manage threads
16       ExecutorService executorService = Executors.newCachedThreadPool();
17
```

Fig. 23.4 | Using an `ExecutorService` to execute `Runnable`s. (Part 1 of 4.)

```
18          // start the three PrintTasks
19          executorService.execute(task1); // start task1
20          executorService.execute(task2); // start task2
21          executorService.execute(task3); // start task3
22
23          // shut down ExecutorService--it decides when to shut down threads
24          executorService.shutdown();
25
26          System.out.printf("Tasks started, main ends.%n%n");
27      }
28  }
```

Fig. 23.4 | Using an `ExecutorService` to execute `Runnable`s. (Part 2 of 4.)

```
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
```

Fig. 23.4 | Using an `ExecutorService` to execute `Runnable`s. (Part 3 of 4.)

```
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. (Part 4 of 4.)

23.3 Creating and Executing Threads with the Executor Framework (cont.)

- Recommended that you use the `Executor` interface to manage the execution of `Runnable` objects for you.
  - Typically creates and manages a group of threads called a thread pool to execute `Runnables`.
- `Executors` can reuse existing threads and can improve performance by optimizing the number of threads.
- `Executor` method `execute` accepts a `Runnable` as an argument.
- An `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.

23.3 Creating and Executing Threads with the Executor Framework (cont.)

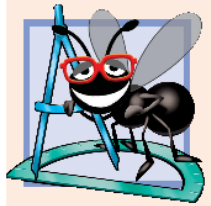
- The `ExecutorService` interface extends `Executor` and declares methods for managing the life cycle of an `Executor`.
- An object that implements this interface can be created using `static` methods declared in class `Executors`.
- `Executors` method `newCachedThreadPool` returns an `ExecutorService` that creates new threads as they're needed by the application.
- `ExecutorService` method `shutdown` notifies the `ExecutorService` to stop accepting new tasks, but continues executing tasks that have already been submitted.

23.4 Thread Synchronization

- When multiple threads share an object and it is *modified* by one or more of them, indeterminate results may occur unless access to the shared object is managed properly.
- 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.
  - Ensures 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

- 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.
- *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

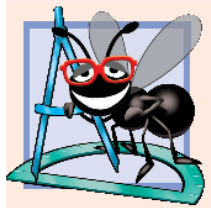
Always declare data fields that you do not expect to change as `final`. Primitive variables that are declared as `final` can safely be shared across threads. An object reference that's declared as `final` ensures that the object it refers to will be fully constructed and initialized before it's used 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).
  - Can be held by a maximum of only one thread at any time.
  - 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.*
- 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.
  - Said to be guarded by the monitor lock

23.4.2 Monitors (Cont.)

- 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.
- When a `synchronized` statement finishes executing, the object's monitor lock is released.
- Java also allows `synchronized` methods.



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

- A `SimpleArray` object (Fig. 23.5) will be *shared* across multiple threads.
- Will enable those threads to place `int` values into `array`.
- 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.

```
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     private static final SecureRandom generator = new SecureRandom();
8     private final int[] array; // the shared integer array
9     private int writeIndex = 0; // shared index of next element to write
10
11    // construct a SimpleArray of a given size
12    public SimpleArray(int size) {array = new int[size];}
13
```

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 3.)

```
14      // add a value to the shared array
15      public void add(int value) {
16         int position = writeIndex; // store the write index
17
18         try {
19            // put thread to sleep for 0-499 milliseconds
20            Thread.sleep(generator.nextInt(500));
21         }
22         catch (InterruptedException ex) {
23            Thread.currentThread().interrupt(); // re-interrupt the thread
24         }
25
26         // put value in the appropriate element
27         array[position] = value;
28         System.out.printf("%s wrote %2d to element %d.%n",
29            Thread.currentThread().getName(), value, position);
30
31         ++writeIndex; // increment index of element to be written next
32         System.out.printf("Next write index: %d%n", writeIndex);
33      }
```

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 3.)

```
34
35      // used for outputting the contents of the shared integer array
36      @Override
37      public String toString() {
38         return Arrays.toString(array);
39      }
40   }
```

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 3 of 3.)

23.4.3 Unsynchronized Data Sharing (cont.)

- Class `ArrayWriter` (Fig. 23.6) implements the interface `Runnable` to define a task for inserting values in a `SimpleArray` object.
- The task completes after three consecutive integers beginning with `startValue` are inserted in the `SimpleArray` object.

```
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     private final SimpleArray sharedSimpleArray;
7     private final int startValue;
8
9     public ArrayWriter(int value, SimpleArray array) {
10       startValue = value;
11       sharedSimpleArray = array;
12    }
13
14    @Override
15    public void run() {
16       for (int i = startValue; i < startValue + 3; i++) {
17          sharedSimpleArray.add(i); // add an element to the shared array
18       }
19    }
20 }
```

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.)

23.4.3 Unsynchronized Data Sharing (cont.)

- Class `SharedArrayTest` (Fig. 23.7) executes two `ArrayWriter` tasks that add values to a single `SimpleArray` object.
- `ExecutorService`'s `shutDown` method prevents additional tasks from starting and to enable the application to terminate when the currently executing tasks complete execution.
- 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—returns control to its caller either when all tasks executing in the `ExecutorService` complete or when the specified timeout elapses.

```
1  // Fig. 23.7: SharedArrayTest.java
2  // Executing two Runnables to add elements to a shared SimpleArray.
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.ExecutorService;
5  import java.util.concurrent.TimeUnit;
6
7  public class SharedArrayTest {
8     public static void main(String[] arg) {
9        // construct the shared object
10       SimpleArray sharedSimpleArray = new SimpleArray(6);
11
12       // create two tasks to write to the shared SimpleArray
13       ArrayWriter writer1 = new ArrayWriter(1, sharedSimpleArray);
14       ArrayWriter writer2 = new ArrayWriter(11, sharedSimpleArray);
15
16       // execute the tasks with an ExecutorService
17       ExecutorService executorService = Executors.newCachedThreadPool();
18       executorService.execute(writer1);
19       executorService.execute(writer2);
20
21       executorService.shutdown();
```

Fig. 23.7 | Executing two `Runnables` to add elements to a shared array—the italicized text is our commentary, which is not part of the program's output. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 1 of 3.)

```
22
23          try {
24             // wait 1 minute for both writers to finish executing
25             boolean tasksEnded =
26                executorService.awaitTermination(1, TimeUnit.MINUTES);
27
28             if (tasksEnded) {
29                System.out.printf("%nContents of SimpleArray:%n");
30                System.out.println(sharedSimpleArray); // print contents
31             }
32             else {
33                System.out.println(
34                   "Timed out while waiting for tasks to finish.");
35             }
36          }
37          catch (InterruptedException ex) {
38             ex.printStackTrace();
39          }
40       }
41    }
```

Fig. 23.7 | Executing two `Runnable`s to add elements to a shared array—the italicized text is our commentary, which is not part of the program's output. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 2 of 3.)

```
pool-1-thread-1 wrote  1 to element 0. — 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. — pool-1-thread-2 overwrote element 0's value
Next write index: 4
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]
```

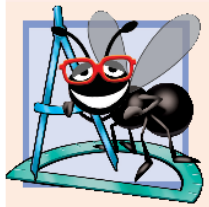
Fig. 23.7 | Executing two `Runnable`s to add elements to a shared array—the italicized text is our commentary, which is not part of the program's output. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 3 of 3.)

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.
- If one thread obtains the value of `writeIndex`, there is 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 is no longer valid.

23.4.4 Synchronized Mutable Data Sharing—Making Operations Atomic (cont.)

- An atomic operation cannot be divided into smaller suboperations.
- Can simulate atomicity by ensuring that only one thread carries out the three operations at a time.
- Atomicity can be achieved using the `synchronized` keyword.



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.

23.4.4 Synchronized Data Sharing—Making Operations Atomic (cont.)

- Figure 23.8 displays class `SimpleArray` with the proper synchronization.
- Identical to the `SimpleArray` class of Fig. 23.5, except that `add` is now a `synchronized` method—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.
- We output messages from `synchronized` blocks for demonstration purposes
  - I/O *should not* be performed in `synchronized` blocks, because it's important to minimize the amount of time that an object is "locked."
- We call `Thread` method `sleep` to emphasize the unpredictability of thread scheduling.
  - *Never* call `sleep` while holding a lock in a real application.

```
1  // Fig. 23.8: SimpleArray.java
2  // Class that manages an integer array to be shared by multiple
3  // threads with synchronization.
4  import java.security.SecureRandom;
5  import java.util.Arrays;
6
7  public class SimpleArray {
8     private static final SecureRandom generator = new SecureRandom();
9     private final int[] array; // the shared integer array
10    private int writeIndex = 0; // index of next element to be written
11
12    // construct a SimpleArray of a given size
13    public SimpleArray(int size) {
14       array = new int[size];
15    }
16
```

Fig. 23.8 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 1 of 4.)

```
17      // add a value to the shared array
18      public synchronized void add(int value) {
19         int position = writeIndex; // store the write index
20
21         try {
22            // in real applications, you shouldn't sleep while holding a lock
23            Thread.sleep(generator.nextInt(500)); // for demo only
24         }
25         catch (InterruptedException ex) {
26            Thread.currentThread().interrupt();
27         }
28
29         // put value in the appropriate element
30         array[position] = value;
31         System.out.printf("%s wrote %2d to element %d.%n",
32            Thread.currentThread().getName(), value, position);
33
34         ++writeIndex; // increment index of element to be written next
35         System.out.printf("Next write index: %d%n", writeIndex);
36      }
```

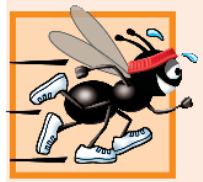
Fig. 23.8 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 2 of 4.)

```
37
38      // used for outputting the contents of the shared integer array
39      @Override
40      public synchronized String toString() {
41         return Arrays.toString(array);
42      }
43   }
```

Fig. 23.8 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 3 of 4.)

```
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 4 of 4.)



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.*
- 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.
- 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.

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

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.)

```
1  // Fig. 23.10: Producer.java
2  // Producer with a run method that inserts the values 1 to 10 in buffer.
3  import java.security.SecureRandom;
4
5  public class Producer implements Runnable {
6     private static final SecureRandom generator = new SecureRandom();
7     private final Buffer sharedLocation; // reference to shared object
8
9     // constructor
10    public Producer(Buffer sharedLocation) {
11       this.sharedLocation = sharedLocation;
12    }
13
```

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.) (Part 1 of 2.)

```
14      // store values from 1 to 10 in sharedLocation
15      @Override
16      public void run() {
17         int sum = 0;
18
19         for (int count = 1; count <= 10; count++) {
20            try { // sleep 0 to 3 seconds, then place value in Buffer
21               Thread.sleep(generator.nextInt(3000)); // random sleep
22               sharedLocation.blockingPut(count); // set value in buffer
23               sum += count; // increment sum of values
24               System.out.printf("\t%2d%n", sum);
25            }
26            catch (InterruptedException exception) {
27               Thread.currentThread().interrupt();
28            }
29         }
30
31         System.out.printf(
32            "Producer done producing%nTerminating Producer%n");
33      }
34   }
```

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.) (Part 2 of 2.)

```
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     private static final SecureRandom generator = new SecureRandom();
7     private final Buffer sharedLocation; // reference to shared object
8
9     // constructor
10    public Consumer(Buffer sharedLocation) {
11       this.sharedLocation = sharedLocation;
12    }
13
```

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.) (Part 1 of 2.)

```
14      // read sharedLocation's value 10 times and sum the values
15      @Override
16      public void run() {
17         int sum = 0;
18
19         for (int count = 1; count <= 10; count++) {
20            // sleep 0 to 3 seconds, read value from buffer and add to sum
21            try {
22               Thread.sleep(generator.nextInt(3000));
23               sum += sharedLocation.blockingGet();
24               System.out.printf("\t\t\t%2d%n", sum);
25            }
26            catch (InterruptedException exception) {
27               Thread.currentThread().interrupt();
28            }
29         }
30
31         System.out.printf("%n%s %d%n%s%n",
32            "Consumer read values totaling", sum, "Terminating Consumer");
33      }
34   }
```

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.) (Part 2 of 2.)

```
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     private int buffer = -1; // shared by producer and consumer threads
6
7     // place value into buffer
8     @Override
9     public void blockingPut(int value) throws InterruptedException {
10       System.out.printf("Producer writes\t%2d", value);
11       buffer = value;
12    }
13
14    // return value from buffer
15    @Override
16    public int blockingGet() throws InterruptedException {
17       System.out.printf("Consumer reads\t%2d", buffer);
18       return buffer;
19    }
20 }
```

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.)

```
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     public static void main(String[] args) throws InterruptedException {
9        // create new thread pool
10       ExecutorService executorService = Executors.newCachedThreadPool();
11
12       // create UnsynchronizedBuffer to store ints
13       Buffer sharedLocation = new UnsynchronizedBuffer();
14
15       System.out.println(
16          "Action\t\tValue\tSum of Produced\tSum of Consumed");
17       System.out.printf(
18          "------\t\t-----\t---------------\t---------------%n%n");
```

Fig. 23.13 | Application with two threads manipulating an unsynchronized buffer—the italicized text in the output is our commentary, which is not part of the program's output. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.) (Part 1 of 6.)

```
19
20          // execute the Producer and Consumer, giving each
21          // access to the sharedLocation
22          executorService.execute(new Producer(sharedLocation));
23          executorService.execute(new Consumer(sharedLocation));
24
25          executorService.shutdown(); // terminate app when tasks complete
26          executorService.awaitTermination(1, TimeUnit.MINUTES);
27       }
28    }
```

Fig. 23.13 | Application with two threads manipulating an unsynchronized buffer—the italicized text in the output is our commentary, which is not part of the program's output. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.) (Part 2 of 6.)

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

Producer writes  1        1
Producer writes  2        3                             — 1 lost
Producer writes  3        6                             — 2 lost
Consumer reads   3                        3
Producer writes  4       10
Consumer reads   4                        7
Producer writes  5       15
Producer writes  6       21                             — 5 lost
Producer writes  7       28                             — 6 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 lost
```

Fig. 23.13 | Application with two threads manipulating an unsynchronized buffer—the italicized text in the output is our commentary, which is not part of the program's output. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.) (Part 3 of 6.)

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

Consumer read values totaling 77
Terminating Consumer
```

Fig. 23.13 | Application with two threads manipulating an unsynchronized buffer—the italicized text in the output is our commentary, which is not part of the program's output. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.) (Part 4 of 6.)

```
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 lost
Consumer reads   6                            19
```

Fig. 23.13 | Application with two threads manipulating an unsynchronized buffer—the italicized text in the output is our commentary, which is not part of the program's output. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.) (Part 5 of 6.)

```
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                  — 9 never read

Producer done producing
Terminating Producer
```

Fig. 23.13 | Application with two threads manipulating an unsynchronized buffer—the italicized text in the output is our commentary, which is not part of the program's output. (*Caution:* The example of Figs. 23.9–23.13 is *not* thread safe.) (Part 6 of 6.)

Error-Prevention Tip 23.1

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

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-` (from package `java.util.concurrent`)—a fully implemented, *thread-safe buffer class* that implements interface `BlockingQueue`.
- 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.

```
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     private final ArrayBlockingQueue<Integer> buffer; // shared buffer
7
8     public BlockingBuffer() {
9        buffer = new ArrayBlockingQueue<Integer>(1);
10    }
11
12    // place value into buffer
13    @Override
14    public void blockingPut(int value) throws InterruptedException {
15       buffer.put(value); // place value in buffer
16       System.out.printf("%s%2d\t%s%d%n", "Producer writes ", value,
17          "Buffer cells occupied: ", buffer.size());
18    }
19
```

Fig. 23.14 | Creating a synchronized buffer using an `ArrayBlockingQueue`. (Part 1 of 2.)

```
20        // return value from buffer
21        @Override
22        public int blockingGet() throws InterruptedException {
23           int readValue = buffer.take(); // remove value from buffer
24           System.out.printf("%s %2d\t%s%d%n", "Consumer reads ",
25              readValue, "Buffer cells occupied: ", buffer.size());
26
27           return readValue;
28        }
29     }
```

Fig. 23.14 | Creating a synchronized buffer using an `ArrayBlockingQueue`. (Part 2 of 2.)

```
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     public static void main(String[] args) throws InterruptedException {
10       // create new thread pool
11       ExecutorService executorService = Executors.newCachedThreadPool();
12
13       // create BlockingBuffer to store ints
14       Buffer sharedLocation = new BlockingBuffer();
15
16       executorService.execute(new Producer(sharedLocation));
17       executorService.execute(new Consumer(sharedLocation));
18
19       executorService.shutdown();
20       executorService.awaitTermination(1, TimeUnit.MINUTES);
21    }
22 }
```

Fig. 23.15 | Two threads manipulating a blocking buffer that properly implements the producer/consumer relationship. (Part 1 of 3.)

```
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
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
```

Fig. 23.15 | Two threads manipulating a blocking buffer that properly implements the producer/consumer relationship. (Part 2 of 3.)

```
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 3 of 3.)

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

- For educational purposes, we now explain how you can implement a shared buffer yourself using the `synchronized` keyword and methods of class `Object`.
- 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.

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

- `Object` methods `wait`, `notify` and `notifyAll` can be used with conditions to make threads wait when they cannot perform their tasks.
- Calling `Object` method `wait` on a `synchronized` object releases its *monitor lock*, and places the calling thread in the *waiting* state.
- Call `Object` method `notify` on a `synchronized` object allows a waiting thread to transition to the *runnable state* again.
- If a thread calls `notifyAll` on the `synchronized` object, then all the threads waiting for the monitor lock become eligible to reacquire the lock.

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.

```
1  // Fig. 23.16: SynchronizedBuffer.java
2  // Synchronizing access to shared mutable data using Object
3  // methods wait and notifyAll.
4  public class SynchronizedBuffer implements Buffer {
5     private int buffer = -1; // shared by producer and consumer threads
6     private boolean occupied = false;
7
```

Fig. 23.16 | Synchronizing access to shared mutable data using `Object` methods `wait` and `notifyAll`. (Part 1 of 4.)

```
8       // place value into buffer
9       @Override
10      public synchronized void blockingPut(int value)
11         throws InterruptedException {
12         // while there are no empty locations, place thread in waiting state
13         while (occupied) {
14            // output thread information and buffer information, then wait
15            System.out.println("Producer tries to write."); // for demo only
16            displayState("Buffer full. Producer waits."); // for demo only
17            wait();
18         }
19
20         buffer = value; // set new buffer value
21
22         // indicate producer cannot store another value
23         // until consumer retrieves current buffer value
24         occupied = true;
25
26         displayState("Producer writes " + buffer); // for demo only
27
28         notifyAll(); // tell waiting thread(s) to enter runnable state
29      } // end method blockingPut; releases lock on SynchronizedBuffer
```

Fig. 23.16 | Synchronizing access to shared mutable data using `Object` methods `wait` and `notifyAll`. (Part 2 of 4.)

```
30
31      // return value from buffer
32      @Override
33      public synchronized int blockingGet() throws InterruptedException {
34         // while no data to read, place thread in waiting state
35         while (!occupied) {
36            // output thread information and buffer information, then wait
37            System.out.println("Consumer tries to read."); // for demo only
38            displayState("Buffer empty. Consumer waits."); // for demo only
39            wait();
40         }
41
42         // indicate that producer can store another value
43         // because consumer just retrieved buffer value
44         occupied = false;
45
46         displayState("Consumer reads " + buffer); // for demo only
47
48         notifyAll(); // tell waiting thread(s) to enter runnable state
49
50         return buffer;
51      } // end method blockingGet; releases lock on SynchronizedBuffer
```

Fig. 23.16 | Synchronizing access to shared mutable data using `Object` methods `wait` and `notifyAll`. (Part 3 of 4.)

```
52
53      // display current operation and buffer state; for demo only
54      private synchronized void displayState(String operation) {
55         System.out.printf("%-40s%d\t\t%b%n%n", operation, buffer, occupied);
56      }
57   }
```

Fig. 23.16 | Synchronizing access to shared mutable data using `Object` methods `wait` and `notifyAll`. (Part 4 of 4.)

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

- The synchronization is handled in class `SynchronizedBuffer`'s `blockingPut` and `blockingGet` methods (Fig. 23.16), which implements interface `Buffer`
- Thus, the `Producer`'s and `Consumer`'s `run` methods simply call the shared object's `synchronized blockingPut` and `blockingGet` methods.

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.

```
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     public static void main(String[] args) throws InterruptedException {
9        // create a newCachedThreadPool
10       ExecutorService executorService = Executors.newCachedThreadPool();
11
12       // create SynchronizedBuffer to store ints
13       Buffer sharedLocation = new SynchronizedBuffer();
14
```

Fig. 23.17 | Two threads correctly manipulating a synchronized buffer. (Part 1 of 6.)

```
15          System.out.printf("%-40s%s\t\t%s%n%-40s%s%n%n", "Operation",
16             "Buffer", "Occupied", "---------", "------\t\t--------");
17
18          // execute the Producer and Consumer tasks
19          executorService.execute(new Producer(sharedLocation));
20          executorService.execute(new Consumer(sharedLocation));
21
22          executorService.shutdown();
23          executorService.awaitTermination(1, TimeUnit.MINUTES);
24       }
25    }
```

Fig. 23.17 | Two threads correctly manipulating a synchronized buffer. (Part 2 of 6.)

```
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
```

Fig. 23.17 | Two threads correctly manipulating a synchronized buffer. (Part 3 of 6.)

```
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
```

Fig. 23.17 | Two threads correctly manipulating a synchronized buffer. (Part 4 of 6.)

```
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
```

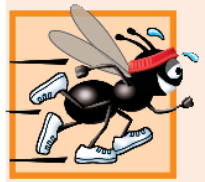
Fig. 23.17 | Two threads correctly manipulating a synchronized buffer. (Part 5 of 6.)

```
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 6 of 6.)

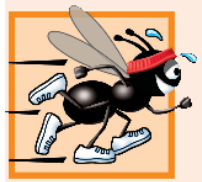
23.8 (Advanced) Producer/Consumer Relationship: Bounded Buffers

- The program in Section 23.7 may not perform optimally.
- If the two threads operate at different speeds, one them will spend more (or most) of its time waiting.
- 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.
- 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.



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.



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.

23.8 (Advanced) Producer/Consumer Relationship: Bounded Buffers (cont.)

- 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 reusing the example from Section 23.6 and simply passing the desired size for the bounded buffer into the `ArrayBlockingQueue` constructor.

23.8 (Advanced) Producer/Consumer Relationship: Bounded Buffers (cont.)

- Implementing Your Own Bounded Buffer as a Circular Buffer
  - The program in Fig. 23.18 and Fig. 23.19 demonstrates a `Producer` and a `Consumer` accessing a bounded buffer with synchronization.
- We implement the bounded buffer in class `CircularBuffer` (Fig. 23.18) as a circular buffer that 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.

```
1  // Fig. 23.18: CircularBuffer.java
2  // Synchronizing access to a shared three-element bounded buffer.
3  public class CircularBuffer implements Buffer {
4     private final int[] buffer = {-1, -1, -1}; // shared buffer
5
6     private int occupiedCells = 0; // count number of buffers used
7     private int writeIndex = 0; // index of next element to write to
8     private int readIndex = 0; // index of next element to read
9
```

Fig. 23.18 | Synchronizing access to a shared three-element bounded buffer. (Part 1 of 5.)

```
10      // place value into buffer
11      @Override
12      public synchronized void blockingPut(int value)
13         throws InterruptedException {
14
15         // wait until buffer has space available, then write value;
16         // while no empty locations, place thread in blocked state
17         while (occupiedCells == buffer.length) {
18            System.out.printf("Buffer is full. Producer waits.%n");
19            wait(); // wait until a buffer cell is free
20         }
21
22         buffer[writeIndex] = value; // set new buffer value
23
24         // update circular write index
25         writeIndex = (writeIndex + 1) % buffer.length;
26
27         ++occupiedCells; // one more buffer cell is full
28         displayState("Producer writes " + value);
29         notifyAll(); // notify threads waiting to read from buffer
30      }
```

Fig. 23.18 | Synchronizing access to a shared three-element bounded buffer. (Part 2 of 5.)

```
31
32      // return value from buffer
33      @Override
34      public synchronized int blockingGet() throws InterruptedException {
35         // wait until buffer has data, then read value;
36         // while no data to read, place thread in waiting state
37         while (occupiedCells == 0) {
38            System.out.printf("Buffer is empty. Consumer waits.%n");
39            wait(); // wait until a buffer cell is filled
40         }
41
42         int readValue = buffer[readIndex]; // read value from buffer
43
44         // update circular read index
45         readIndex = (readIndex + 1) % buffer.length;
46
47         --occupiedCells; // one fewer buffer cells are occupied
48         displayState("Consumer reads " + readValue);
49         notifyAll(); // notify threads waiting to write to buffer
50
51         return readValue;
52      }
```

Fig. 23.18 | Synchronizing access to a shared three-element bounded buffer. (Part 3 of 5.)

```
53
54      // display current operation and buffer state
55      public synchronized void displayState(String operation) {
56         // output operation and number of occupied buffer cells
57         System.out.printf("%s%s%d)%n%s", operation,
58            " (buffer cells occupied: ", occupiedCells, "buffer cells:  ");
59
60         for (int value : buffer) {
61            System.out.printf(" %2d  ", value); // output values in buffer
62         }
63
64         System.out.printf("%n                ");
65
66         for (int i = 0; i < buffer.length; i++) {
67            System.out.print("---- ");
68         }
69
70         System.out.printf("%n                ");
```

Fig. 23.18 | Synchronizing access to a shared three-element bounded buffer. (Part 4 of 5.)

```
71
72          for (int i = 0; i < buffer.length; i++) {
73             if (i == writeIndex && i == readIndex) {
74                System.out.print(" WR"); // both write and read index
75             }
76             else if (i == writeIndex) {
77                System.out.print(" W   "); // just write index
78             }
79             else if (i == readIndex) {
80                System.out.print("  R  "); // just read index
81             }
82             else {
83                System.out.print("     "); // neither index
84             }
85          }
86
87          System.out.printf("%n%n");
88       }
89    }
```

Fig. 23.18 | Synchronizing access to a shared three-element bounded buffer. (Part 5 of 5.)

```
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     public static void main(String[] args) throws InterruptedException {
9        // create new thread pool
10       ExecutorService executorService = Executors.newCachedThreadPool();
11
12       // create CircularBuffer to store ints
13       CircularBuffer sharedLocation = new CircularBuffer();
14
```

Fig. 23.19 | Producer and Consumer threads correctly manipulating a circular buffer. (Part 1 of 8.)

```
15          // display the initial state of the CircularBuffer
16          sharedLocation.displayState("Initial State");
17
18          // execute the Producer and Consumer tasks
19          executorService.execute(new Producer(sharedLocation));
20          executorService.execute(new Consumer(sharedLocation));
21
22          executorService.shutdown();
23          executorService.awaitTermination(1, TimeUnit.MINUTES);
24       }
25    }
```

Fig. 23.19 | `Producer` and `Consumer` threads correctly manipulating a circular buffer. (Part 2 of 8.)

```
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
```

Fig. 23.19 | Producer and Consumer threads correctly manipulating a circular buffer. (Part 3 of 8.)

```
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
```

Fig. 23.19 | Producer and Consumer threads correctly manipulating a circular buffer. (Part 4 of 8.)

```
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 5 of 8.)

```
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
```

Fig. 23.19 | `Producer` and `Consumer` threads correctly manipulating a circular buffer. (Part 6 of 8.)

```
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
```

Fig. 23.19 | `Producer` and `Consumer` threads correctly manipulating a circular buffer. (Part 7 of 8.)

```
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 8 of 8.)

23.9 (Advanced) Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces

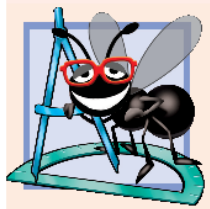
- In this section, we discuss the `Lock` and `Condition` interfaces.
- These interfaces give you more precise control over thread synchronization, but are more complicated to use.
- Any object can contain a reference to an object that implements the `Lock` interface (of package `java.util.concurrent.locks`).
- A thread calls the `Lock`'s `lock` method 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).
- 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.*

23.9 (Advanced) Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces (cont.)

- When a thread calls method `unlock`, the lock on the object is released and a waiting thread attempting to lock the object proceeds.
- Class `ReentrantLock` (of package `java.util.concurrent.locks`) is a basic implementation of the `Lock` interface.
- The constructor for a `ReentrantLock` takes a `boolean` argument that specifies whether the lock has a fairness policy.
- If the argument is `true`, the `ReentrantLock`'s fairness policy is "the longest-waiting thread will acquire the lock when it's available."

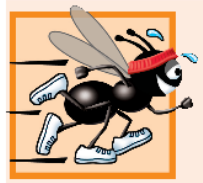
Error-Prevention Tip 23.4

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



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.

23.9 (Advanced) Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces (cont.)

- 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.
  - 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.
  - 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`).

23.9 Producer/Consumer Relationship: The Lock and Condition Interfaces (cont.)

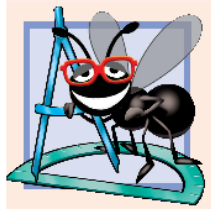
- To wait on a condition object, the thread can call the `Condition`'s `await` method.
  - This immediately releases the associated `Lock` and places the thread in the *waiting state* for that `Condition`.
- When a *runnable* thread completes a task and determines that the *waiting* thread can now continue, the *runnable* thread can call `Condition` method `signal` to allow a thread in that `Condition`'s *waiting state* to return to the *runnable state.*
- 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`.

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.



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`.

23.9 (Advanced) Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces (cont.)

- `Lock`s allow you to interrupt waiting threads or to specify a time-out 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.
- `Condition` objects allow you to specify multiple conditions on which threads may wait.
- With the `synchronized` keyword, there is no way to explicitly state the condition on which threads are waiting.

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.

23.9 (Advanced) Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces (cont.)

- We create two `Conditions` using `Lock` method `newCondition`.
- `Condition canWrite` contains a queue for a `Producer` thread waiting while the buffer is full.
  - 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 is 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`.

```
1  // Fig. 23.20: SynchronizedBuffer.java
2  // Synchronizing access to a shared integer using the Lock and Condition
3  // interfaces
4  import java.util.concurrent.locks.Lock;
5  import java.util.concurrent.locks.ReentrantLock;
6  import java.util.concurrent.locks.Condition;
7
8  public class SynchronizedBuffer implements Buffer {
9     // Lock to control synchronization with this buffer
10    private final Lock accessLock = new ReentrantLock();
11
12    // conditions to control reading and writing
13    private final Condition canWrite = accessLock.newCondition();
14    private final Condition canRead = accessLock.newCondition();
15
16    private int buffer = -1; // shared by producer and consumer threads
17    private boolean occupied = false; // whether buffer is occupied
```

Fig. 23.20 | Synchronizing access to a shared integer using the `Lock` and `Condition` interfaces. (Part 1 of 6.)

```
18
19      // place int value into buffer
20      @Override
21      public void blockingPut(int value) throws InterruptedException {
22         accessLock.lock(); // lock this object
23
24         // output thread information and buffer information, then wait
25         try {
26            // while buffer is not empty, place thread in waiting state
27            while (occupied) {
28               System.out.println("Producer tries to write.");
29               displayState("Buffer full. Producer waits.");
30               canWrite.await(); // wait until buffer is empty
31            }
32
33            buffer = value; // set new buffer value
34
```

Fig. 23.20 | Synchronizing access to a shared integer using the `Lock` and `Condition` interfaces. (Part 2 of 6.)

```
35             // indicate producer cannot store another value
36             // until consumer retrieves current buffer value
37             occupied = true;
38
39             displayState("Producer writes " + buffer);
40
41             // signal any threads waiting to read from buffer
42             canRead.signalAll();
43          }
44          finally {
45             accessLock.unlock(); // unlock this object
46          }
47       }
48
```

Fig. 23.20 | Synchronizing access to a shared integer using the `Lock` and `Condition` interfaces. (Part 3 of 6.)

```
49      // return value from buffer
50      @Override
51      public int blockingGet() throws InterruptedException {
52         int readValue = 0; // initialize value read from buffer
53         accessLock.lock(); // lock this object
54
55         // output thread information and buffer information, then wait
56         try {
57            // if there is no data to read, place thread in waiting state
58            while (!occupied) {
59               System.out.println("Consumer tries to read.");
60               displayState("Buffer empty. Consumer waits.");
61               canRead.await(); // wait until buffer is full
62            }
63
64            // indicate that producer can store another value
65            // because consumer just retrieved buffer value
66            occupied = false;
```

Fig. 23.20 | Synchronizing access to a shared integer using the `Lock` and `Condition` interfaces. (Part 4 of 6.)

```
67
68             readValue = buffer; // retrieve value from buffer
69             displayState("Consumer reads " + readValue);
70
71             // signal any threads waiting for buffer to be empty
72             canWrite.signalAll();
73          }
74          finally {
75             accessLock.unlock(); // unlock this object
76          }
77
78          return readValue;
79       }
80
```

Fig. 23.20 | Synchronizing access to a shared integer using the `Lock` and `Condition` interfaces. (Part 5 of 6.)

```
81      // display current operation and buffer state
82      private void displayState(String operation) {
83         try {
84            accessLock.lock(); // lock this object
85            System.out.printf("%-40s%d\t\t%b%n%n", operation, buffer,
86               occupied);
87         }
88         finally {
89            accessLock.unlock(); // unlock this object
90         }
91      }
92   }
```

Fig. 23.20 | Synchronizing access to a shared integer using the `Lock` and `Condition` interfaces. (Part 6 of 6.)

Common Programming Error 23.3

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

```
1  // Fig. 23.21: SharedBufferTest2.java
2  // Two threads 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     public static void main(String[] args) throws InterruptedException {
9        // create new thread pool
10       ExecutorService executorService = Executors.newCachedThreadPool();
11
12       // create SynchronizedBuffer to store ints
13       Buffer sharedLocation = new SynchronizedBuffer();
14
15       System.out.printf("%-40s%s\t\t%s%n%-40s%s%n%n", "Operation",
16          "Buffer", "Occupied", "---------", "------\t\t--------");
```

Fig. 23.21 | Two threads manipulating a synchronized buffer. (Part 1 of 5.)

```
17
18          // execute the Producer and Consumer tasks
19          executorService.execute(new Producer(sharedLocation));
20          executorService.execute(new Consumer(sharedLocation));
21
22          executorService.shutdown();
23          executorService.awaitTermination(1, TimeUnit.MINUTES);
24       }
25    }
```

Fig. 23.21 | Two threads manipulating a synchronized buffer. (Part 2 of 5.)

```
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
```

Fig. 23.21 | Two threads manipulating a synchronized buffer. (Part 3 of 5.)

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

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
```

Fig. 23.21 | Two threads manipulating a synchronized buffer. (Part 4 of 5.)

```
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 5 of 5.)

23.10 Concurrent Collections

- The collections from the `java.util.concurrent` package are specifically designed and optimized for sharing collections among multiple threads.
- Fig. 23.22 lists the many concurrent collections in package `java.util.concurrent`.
- For more Java SE 7 information, visit
  - http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html
- For Java SE 8 information, visit
  - 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.

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

Collection	Description
`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`.

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

Collection	Description
`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.**

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

Collection	Description
`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.

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

Collection	Description
`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 in JavaFX

- All JavaFX applications have a **single** thread, called the **JavaFX application thread**, to handle interactions with the application's controls
- **Typical interactions** include *rendering controls* 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 JavaFX-**application thread

23.11 Multithreading in JavaFX (cont.)

- ***JavaFX's scene graph is not thread safe****—its nodes cannot be manipulated by multiple threads without the risk of incorrect results that might corrupt the scene graph*
- Thread safety in JavaFX applications is achieved by *ensuring that programs manipulate the scene graph from* ***only the JavaFX application*** *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

23.11 Multithreading in JavaFX (cont.)

- It's acceptable to perform brief tasks on the JavaFX application thread **in sequence with GUI component** manipulations
- If an application must perform a lengthy task in response to a user interaction, the JavaFX application thread cannot render controls or respond to events while the thread is tied up in that task
- This causes the GUI to become unresponsive
- It's preferable to handle long-running tasks in **separate threads**, freeing the JavaFX application thread to continue managing other GUI interactions

23.11 Multithreading in JavaFX (cont.)

Platform Method `runLater`

- JavaFX provides multiple mechanisms for updating the GUI from other threads
- One is to call the `static` method **`runLater`** of class **`Platform`** (package `javafx.application`)
- This method receives a `Runnable` and schedules it on the JavaFX application thread for execution at some point in the future
- Such `Runnables` should perform only small updates, so the GUI remains responsive

23.11 Multithreading in JavaFX (cont.)

Class `Task` and Interface `Worker`

- **Long-running or compute-intensive tasks should be performed on separate worker threads**, not the JavaFX application thread
- Package `javafx.concurrent` provides interface `Worker` and classes `Task` and `ScheduledService` for this purpose
- A `Worker` is a task that should execute using one or more separate threads.

23.11 Multithreading in JavaFX (cont.)

Class `Task` and Interface `Worker`

- A `Task` is a `Worker` that enables you to perform a task in a worker thread and update the GUI from the JavaFX application thread based on the task's results
  - `Task` implements `Runnable`, so a `Task` object can be scheduled to execute in a separate thread
  - Class `Task` also provides methods that are guaranteed to update its properties in the JavaFX application thread
  - Once a `Task` completes, it cannot be restarted
- A `ScheduledService` is a `Worker` that creates and manages a `Task`
  - Can be reset and restarted
  - Can be configured to automatically restart both after successful completion and if it fails due to an exception.

23.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers

- Next example: User enters a number n and the program gets the nth 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 `Task` object to perform the recursive calculation in a worker thread
- The GUI also provides a separate set of components that displays the next Fibonacci number in the sequence with each click of a `Button`
- This set of components performs its short computation directly in the event dispatch thread

23.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers (cont.)

Creating a Task

- Class `FibonacciTask` (Fig. 23.23) extends `Task<Long>` to perform the recursive Fibonacci calculation in a worker thread
- The instance variable `n` represents the Fibonacci number to calculate.
- Overridden `Task` method `call` computes the nth Fibonacci number and returns the result
- The `Task`'s type parameter `Long` determines `call`'s return type
- Inherited `Task` method `updateMessage` updates the `Task`'s `message` property in the JavaFX application thread
- In Fig. 23.25, we'll bind JavaFX controls to `FibonacciTask`'s `message` property to display messages while the task is executing

```
1  // Fig. 23.23: FibonacciTask.java
2  // Task subclass for calculating Fibonacci numbers in the background
3  import javafx.concurrent.Task;
4
5  public class FibonacciTask extends Task<Long> {
6     private final int n; // Fibonacci number to calculate
7
8     // constructor
9     public FibonacciTask(int n) {
10       this.n = n;
11    }
12
```

Fig. 23.23 | Task subclass for calculating Fibonacci numbers in the background. (Part 1 of 2.)

```
13      // long-running code to be run in a worker thread
14      @Override
15      protected Long call() {
16         updateMessage("Calculating...");
17         long result = fibonacci(n);
18         updateMessage("Done calculating.");
19         return result;
20      }
21
22      // recursive method fibonacci; calculates nth Fibonacci number
23      public long fibonacci(long number) {
24         if (number == 0 || number == 1) {
25            return number;
26         }
27         else {
28            return fibonacci(number - 1) + fibonacci(number - 2);
29         }
30      }
31   }
```

Fig. 23.23 | `Task` subclass for calculating Fibonacci numbers in the background. (Part 2 of 2.)

23.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers (cont.)

- When the worker thread enters the *running* state, `FibonacciTask`'s call method begins executing
- First, `updateMessage` updates the `FibonacciTask`'s `message` property, indicating that the task is calculating
- When `fibonacci` returns, we update `FibonacciTask`'s message property again to indicate that the calculation completed, then return the result to the JavaFX application thread

23.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers (cont.)

FibonacciNumbers GUI

- Figure 23.24 shows the app's GUI (defined in `FibonacciNumbers.fxml`) labeled with its **fx:id**s.
- Event handlers
  - `goButtonPressed` is called when the Go `Button` is pressed—this launches the worker thread to calculate a Fibonacci number recursively
  - `nextNumberButtonPressed` is called when the Next Number `Button` is pressed—this calculates the next Fibonacci number in the sequence

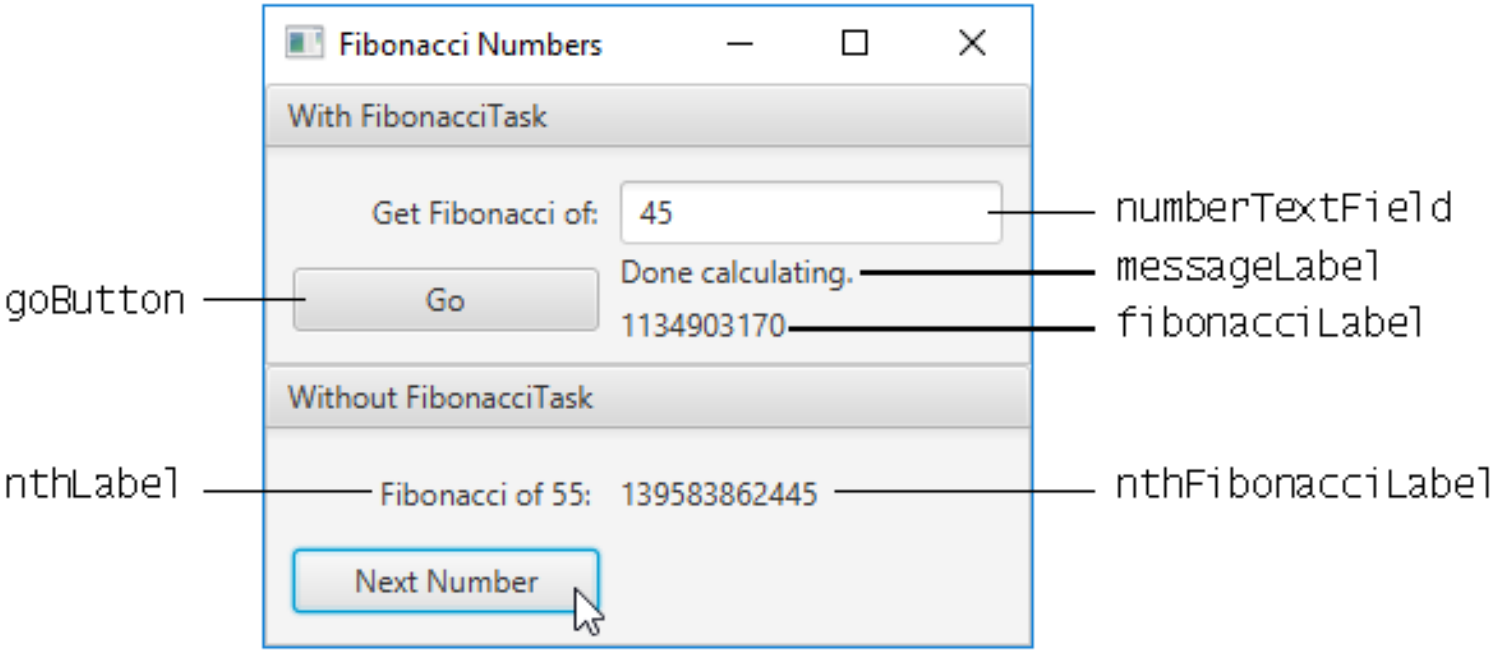


Fig. 23.24 | `FibonacciNumbers` GUI with its **fx:id**s.

23.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers (cont.)

Class FibonacciNumbersController

- Class `FibonacciNumbersController` (Fig. 23.25) displays a window containing two sets of controls:
  - The **With FibonacciTask** `TitledPane` provides controls that enable the user to enter a Fibonacci number to calculate and launch a `FibonacciTask` in a worker thread. `Labels` display the `FibonacciTask`'s `message` property value as it's updated and the final result of the `FibonacciTask` once it's available.
  - The **Without FibonacciTask** `TitledPane` provides the Next Number `Button` that enables the user to calculate the next Fibonacci number in sequence. `Labels` display which Fibonacci number is being calculated and the corresponding Fibonacci value.

23.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers (cont.)

Class FibonacciNumbersController

- Instance variables `n1` and `n2` contain the previous two Fibonacci numbers in the sequence and are initialized to 0 and 1, respectively.
- Instance variable `number` (initialized to 1) keeps track of which Fibonacci value will be calculated and displayed next when the user clicks the **Next Number** `Button`
  - So the first time this `Button` is clicked, the Fibonacci of 1 is displayed.

```
1  // Fig. 23.25: FibonacciNumbersController.java
2  // Using a Task to perform a long calculation
3  // outside the JavaFX application thread.
4  import java.util.concurrent.Executors;
5  import java.util.concurrent.ExecutorService;
6  import javafx.event.ActionEvent;
7  import javafx.fxml.FXML;
8  import javafx.scene.control.Button;
9  import javafx.scene.control.Label;
10 import javafx.scene.control.TextField;
11
12 public class FibonacciNumbersController {
13    @FXML private TextField numberTextField;
14    @FXML private Button goButton;
15    @FXML private Label messageLabel;
16    @FXML private Label fibonacciLabel;
17    @FXML private Label nthLabel;
18    @FXML private Label nthFibonacciLabel;
```

Fig. 23.25 | Using a `Task` to perform a long calculation outside the JavaFX application thread. (Part 1 of 6.)

```
19
20      private long n1 = 0; // initialize with Fibonacci of 0
21      private long n2 = 1; // initialize with Fibonacci of 1
22      private int number = 1; // current Fibonacci number to display
23
24      // starts FibonacciTask to calculate in background
25      @FXML
26      void goButtonPressed(ActionEvent event) {
27         // get Fibonacci number to calculate
28         try {
29            int input = Integer.parseInt(numberTextField.getText());
30
31            // create, configure and launch FibonacciTask
32            FibonacciTask task = new FibonacciTask(input);
33
34            // display task's messages in messageLabel
35            messageLabel.textProperty().bind(task.messageProperty());
36
```

Fig. 23.25 | Using a `Task` to perform a long calculation outside the JavaFX application thread. (Part 2 of 6.)

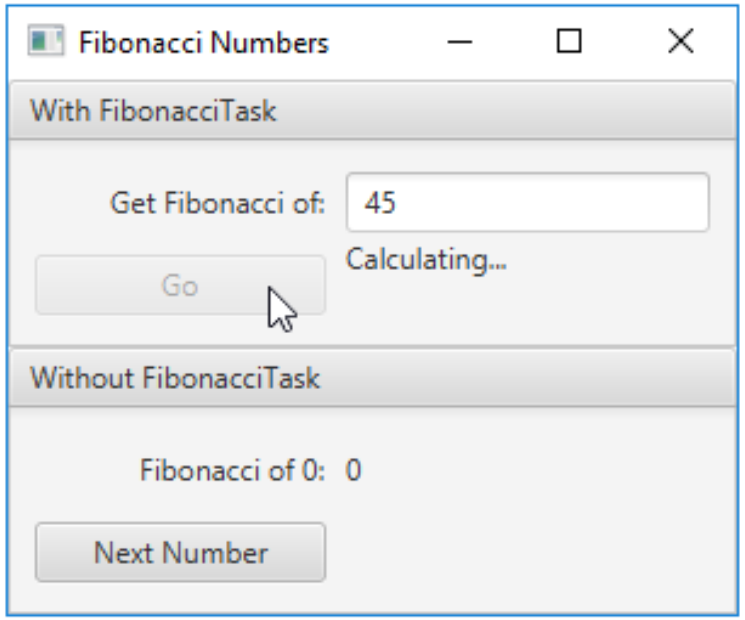
```
37            // clear fibonacciLabel when task starts
38            task.setOnRunning((succeededEvent) -> {
39               goButton.setDisable(true);
40               fibonacciLabel.setText("");
41            });
42
43            // set fibonacciLabel when task completes successfully
44            task.setOnSucceeded((succeededEvent) -> {
45               fibonacciLabel.setText(task.getValue().toString());
46               goButton.setDisable(false);
47            });
48
49            // create ExecutorService to manage threads
50            ExecutorService executorService =
51               Executors.newFixedThreadPool(1); // pool of one thread
52            executorService.execute(task); // start the task
53            executorService.shutdown();
54         }
```

Fig. 23.25 | Using a `Task` to perform a long calculation outside the JavaFX application thread. (Part 3 of 6.)

```
55          catch (NumberFormatException e) {
56             numberTextField.setText("Enter an integer");
57             numberTextField.selectAll();
58             numberTextField.requestFocus();
59          }
60       }
61
62       // calculates next Fibonacci value
63       @FXML
64       void nextNumberButtonPressed(ActionEvent event) {
65          // display the next Fibonacci number
66          nthLabel.setText("Fibonacci of " + number + ": ");
67          nthFibonacciLabel.setText(String.valueOf(n2));
68          long temp = n1 + n2;
69          n1 = n2;
70          n2 = temp;
71          ++number;
72       }
73    }
```

Fig. 23.25 | Using a `Task` to perform a long calculation outside the JavaFX application thread. (Part 4 of 6.)

a) Begin calculating Fibonacci of 45 in the background



b) Calculating other Fibonacci values while Fibonacci of 45 continues calculating

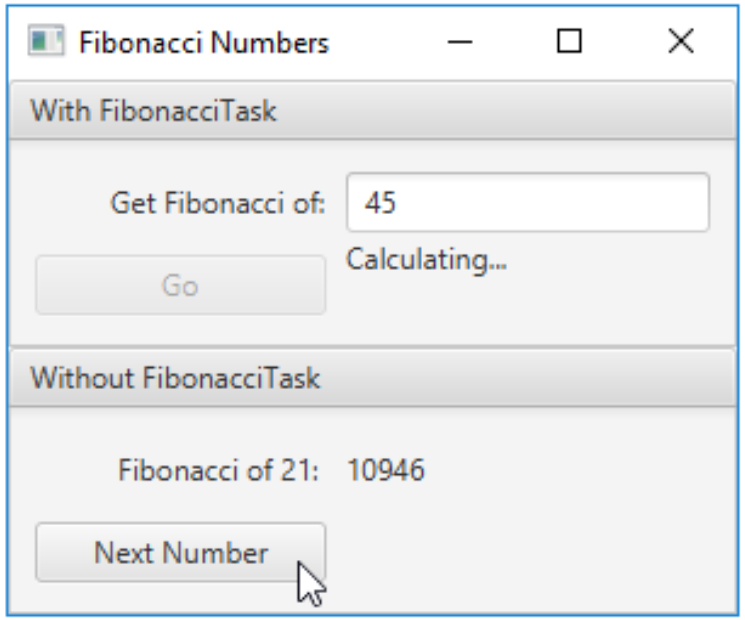


Fig. 23.25 | Using a **Task** to perform a long calculation outside the JavaFX application thread. (Part 5 of 6.)

c) Fibonacci of 45 calculation finishes

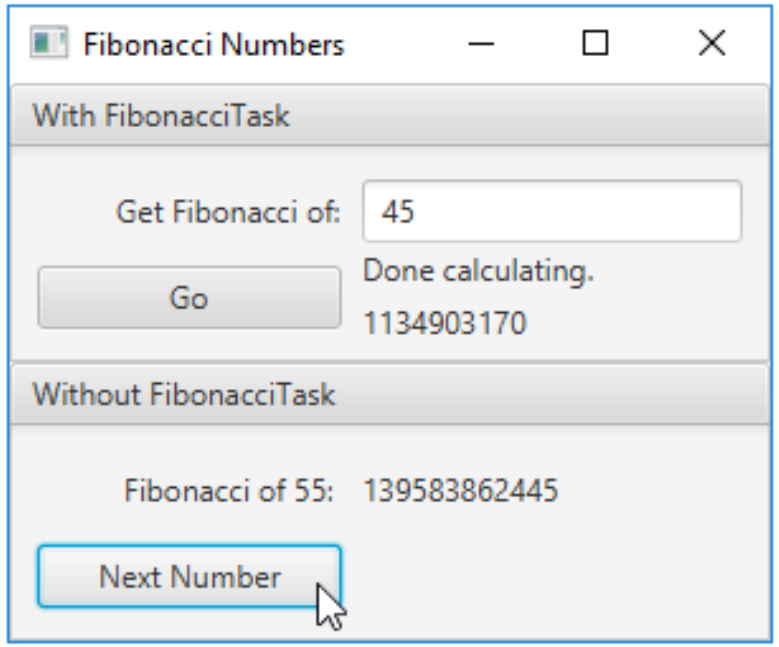


Fig. 23.25 | Using a **Task** to perform a long calculation outside the JavaFX application thread. (Part 6 of 6.)

23.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers (cont.)

Method goButtonPressed

- Gets the value entered in the `numberTextField`
- Creates a new `FibonacciTask` object, passing the constructor the user-entered value
- Binds the `messageLabel`'s `text` property (a `StringProperty`) to the `FibonacciTask`'s `message` property (a `ReadOnlyStringProperty`)
- All `Workers` transition through various states. Class `Task`—an implementation of `Worker`—enables you to register listeners for several of these states:

23.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers (cont.)

Method goButtonPressed

- All `Workers` transition through various states. Class `Task`—an implementation of `Worker`—enables you to register listeners for several of these states:
  - `Task` method `setOnRunning` registers a listener that's invoked when the `Task` enters the running state—that is, when the `Task` has been assigned a processor and begins executing its call method.
  - `Task` method `setOnSucceeded` registers a listener that's invoked when the `Task` enters the succeeded state—that is, when the `Task` successfully runs to completion.

23.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers (cont.)

- You also can register listeners for a `Task`'s *canceled*, *failed* and *scheduled* states.
- We use an `ExecutorService` to launch the `FibonacciTask`, which schedules it for execution in a separate worker thread
- Method `execute` does not wait for the `FibonacciTask` to finish executing—it returns immediately, allowing the GUI to continue processing other events while the computation is performed

23.11.1 Performing Computations in a Worker Thread: Fibonacci Numbers (cont.)

Method nextNumberButtonPressed

- Updates the `nthLabel` to show which Fibonacci number is being displayed, then updates `nthFibonacciLabel` to display `n2`'s value
- Adds the previous two Fibonacci numbers stored in `n1` and `n2` to determine the next number in the sequence, then updates `n1` and `n2` to their new values and increment number.

23.11.2 Processing Intermediate Results: Sieve of Eratosthenes

- Class `Task` provides additional methods and properties that enable you to update a GUI with a `Task`'s intermediate results as the `Task` continues executing in a `Worker` thread.
  - Method `updateProgress` updates a `Task`'s `progress` property, which represents the percentage completion.
  - Method `updateValue` updates a `Task`'s `value` property, which holds each intermediate value.
- Both guaranteed to update the corresponding properties in the JavaFX application thread

23.11.2 Processing Intermediate Results: Sieve of Eratosthenes

- Class `Task` provides additional methods and properties that enable you to update a GUI with a `Task`'s intermediate results as the `Task` continues executing in a `Worker` thread.
  - Method `updateProgress` updates a `Task`'s `progress` property, which represents the percentage completion.
  - Method `updateValue` updates a `Task`'s `value` property, which holds each intermediate value.
- Both guaranteed to update the corresponding properties in the JavaFX application thread

23.11.2 Processing Intermediate Results: Sieve of Eratosthenes (cont.)

- Figure 23.26 presents class `PrimeCalculatorTask`, which extends `Task<Integer>` to compute the first *n* prime numbers in a worker thread
- We'll bind this `Task`'s `progress` property to a `ProgressBar` control so the app can provide a visual indication of the portion of the `Task` that has been completed
- The controller also registers a listener for the `value` property's changes—we'll store each prime value in an `ObservableList` that's bound to a `ListView`.

```
1  // Fig. 23.26: PrimeCalculatorTask.java
2  // Calculates the first n primes, publishing them as they are found.
3  import java.util.Arrays;
4  import javafx.concurrent.Task;
5
6  public class PrimeCalculatorTask extends Task<Integer> {
7     private final boolean[] primes; // boolean array for finding primes
8
9     // constructor
10    public PrimeCalculatorTask(int max) {
11       primes = new boolean[max];
12       Arrays.fill(primes, true); // initialize all primes elements to true
13    }
14
```

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

```
15      // long-running code to be run in a worker thread
16      @Override
17      protected Integer call() {
18         int count = 0; // the number of primes found
19
20         // starting at index 2 (the first prime number), cycle through and
21         // set to false elements with indices that are multiples of i
22         for (int i = 2; i < primes.length; i++) {
23            if (isCancelled()) { // if calculation has been canceled
24               updateMessage("Cancelled");
25               return 0;
26            }
27            else {
28               try {
29                  Thread.sleep(10); // slow the thread
30               }
31               catch (InterruptedException ex) {
32                  updateMessage("Interrupted");
33                  return 0;
34               }
```

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

```
35
36                updateProgress(i + 1, primes.length);
37
38                if (primes[i]) { // i is prime
39                   ++count;
40                   updateMessage(String.format("Found %d primes", count));
41                   updateValue(i); // intermediate result
42
43                   // eliminate multiples of i
44                   for (int j = i + i; j < primes.length; j += i) {
45                      primes[j] = false; // i is not prime
46                   }
47                }
48             }
49          }
50
51          return 0;
52       }
53    }
```

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

23.11.2 Processing Intermediate Results: Sieve of Eratosthenes (cont.)

Overridden Task Method call

- In method `call`, the control variable `i` for the loop represents the current index for implementing the Sieve of Eratosthenes
- `Task` method **`isCancelled`** determines whether the user has clicked the **Cancel** button
  - If so, we update the `Task`'s `message` property, then return 0 to terminate the Task immediately

23.11.2 Processing Intermediate Results: Sieve of Eratosthenes (cont.)

Overridden Task Method call

- If the calculation isn't canceled, we put the currently executing thread to sleep for 10 milliseconds
  - Because the computation progresses quickly, publishing values often, updates can pile up on the JavaFX application thread, causing it to ignore some updates
  - This is why for demonstration purposes we put the worker thread to *sleep* for 10 milliseconds during each iteration of the loop.
  - The calculation is slowed just enough to allow the JavaFX application thread to keep up with the updates and enable the GUI to remain responsive

23.11.2 Processing Intermediate Results: Sieve of Eratosthenes (cont.)

Overridden Task Method call

- `Task`'s `updateProgress` method updates the `progress` property in the JavaFX application thread
- Next, we test whether the current primes element is true (and thus prime). If so, we increment the count of prime numbers found so far and update `Task`'s `message` property with a `String` containing the count
- Then, we passes the index to method `updateValue`, which updates `Task`'s `value` property in the JavaFX application thread

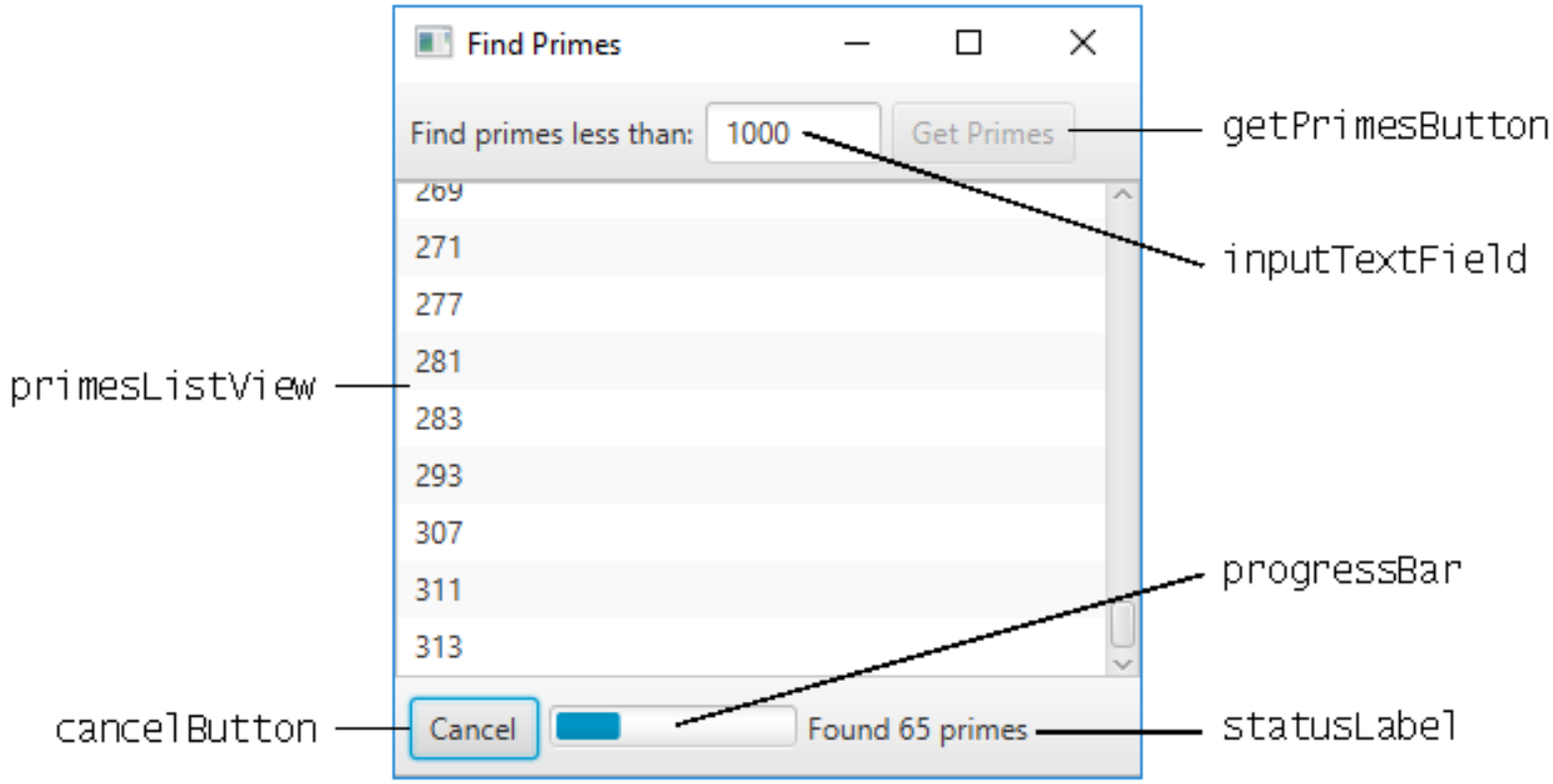


Fig. 23.27 | **FindPrimes** GUI with its **fx:id**s.

23.11.2 Processing Intermediate Results: Sieve of Eratosthenes (cont.)

Class FindPrimesController

- Class `FindPrimesController` (Fig. 23.28) creates an `ObservableList<Integer>` and binds it to the `primesListView`
- The controller also provides the event handlers for the **Get Primes** and **Cancel** `Button`s

```
1  // Fig. 23.28: FindPrimesController.java
2  // Displaying prime numbers as they're calculated; updating a ProgressBar
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.ExecutorService;
5  import javafx.collections.FXCollections;
6  import javafx.collections.ObservableList;
7  import javafx.event.ActionEvent;
8  import javafx.fxml.FXML;
9  import javafx.scene.control.Button;
10 import javafx.scene.control.Label;
11 import javafx.scene.control.ListView;
12 import javafx.scene.control.ProgressBar;
13 import javafx.scene.control.TextField;
14
15 public class FindPrimesController {
16    @FXML private TextField inputTextField;
17    @FXML private Button getPrimesButton;
18    @FXML private ListView<Integer> primesListView;
19    @FXML private Button cancelButton;
20    @FXML private ProgressBar progressBar;
21    @FXML private Label statusLabel;
```

Fig. 23.28 | Displaying prime numbers as they're calculated and updating a `ProgressBar`. (Part 1 of 7.)

```
22
23      // stores the list of primes received from PrimeCalculatorTask
24      private ObservableList<Integer> primes =
25         FXCollections.observableArrayList();
26      private PrimeCalculatorTask task; // finds prime numbers
27
28      // binds primesListView's items to the ObservableList primes
29      public void initialize() {
30         primesListView.setItems(primes);
31      }
32
33      // start calculating primes in the background
34      @FXML
35      void getPrimesButtonPressed(ActionEvent event) {
36         primes.clear();
37
38         // get Fibonacci number to calculate
39         try {
40            int input = Integer.parseInt(inputTextField.getText());
41            task = new PrimeCalculatorTask(input); // create task
```

Fig. 23.28 | Displaying prime numbers as they're calculated and updating a `ProgressBar`. (Part 2 of 7.)

```
42
43              // display task's messages in statusLabel
44              statusLabel.textProperty().bind(task.messageProperty());
45
46              // update progressBar based on task's progressProperty
47              progressBar.progressProperty().bind(task.progressProperty());
48
49              // store intermediate results in the ObservableList primes
50              task.valueProperty().addListener(
51                 (observable, oldValue, newValue) -> {
52                    if (newValue != 0) { // task returns 0 when it terminates
53                       primes.add(newValue);
54                       primesListView.scrollTo(
55                          primesListView.getItems().size());
56                    }
57                 });
58
```

Fig. 23.28 | Displaying prime numbers as they're calculated and updating a `ProgressBar`. (Part 3 of 7.)

```
59            // when task begins,
60            // disable getPrimesButton and enable cancelButton
61            task.setOnRunning((succeededEvent) -> {
62               getPrimesButton.setDisable(true);
63               cancelButton.setDisable(false);
64            });
65
66            // when task completes successfully,
67            // enable getPrimesButton and disable cancelButton
68            task.setOnSucceeded((succeededEvent) -> {
69               getPrimesButton.setDisable(false);
70               cancelButton.setDisable(true);
71            });
72
73            // create ExecutorService to manage threads
74            ExecutorService executorService =
75               Executors.newFixedThreadPool(1);
76            executorService.execute(task); // start the task
77            executorService.shutdown();
78         }
```

Fig. 23.28 | Displaying prime numbers as they're calculated and updating a `ProgressBar`. (Part 4 of 7.)

```
79          catch (NumberFormatException e) {
80             inputTextField.setText("Enter an integer");
81             inputTextField.selectAll();
82             inputTextField.requestFocus();
83          }
84       }
85
86       // cancel task when user presses Cancel Button
87       @FXML
88       void cancelButtonPressed(ActionEvent event) {
89          if (task != null) {
90             task.cancel(); // terminate the task
91             getPrimesButton.setDisable(false);
92             cancelButton.setDisable(true);
93          }
94       }
95    }
```

Fig. 23.28 | Displaying prime numbers as they're calculated and updating a `ProgressBar`. (Part 5 of 7.)

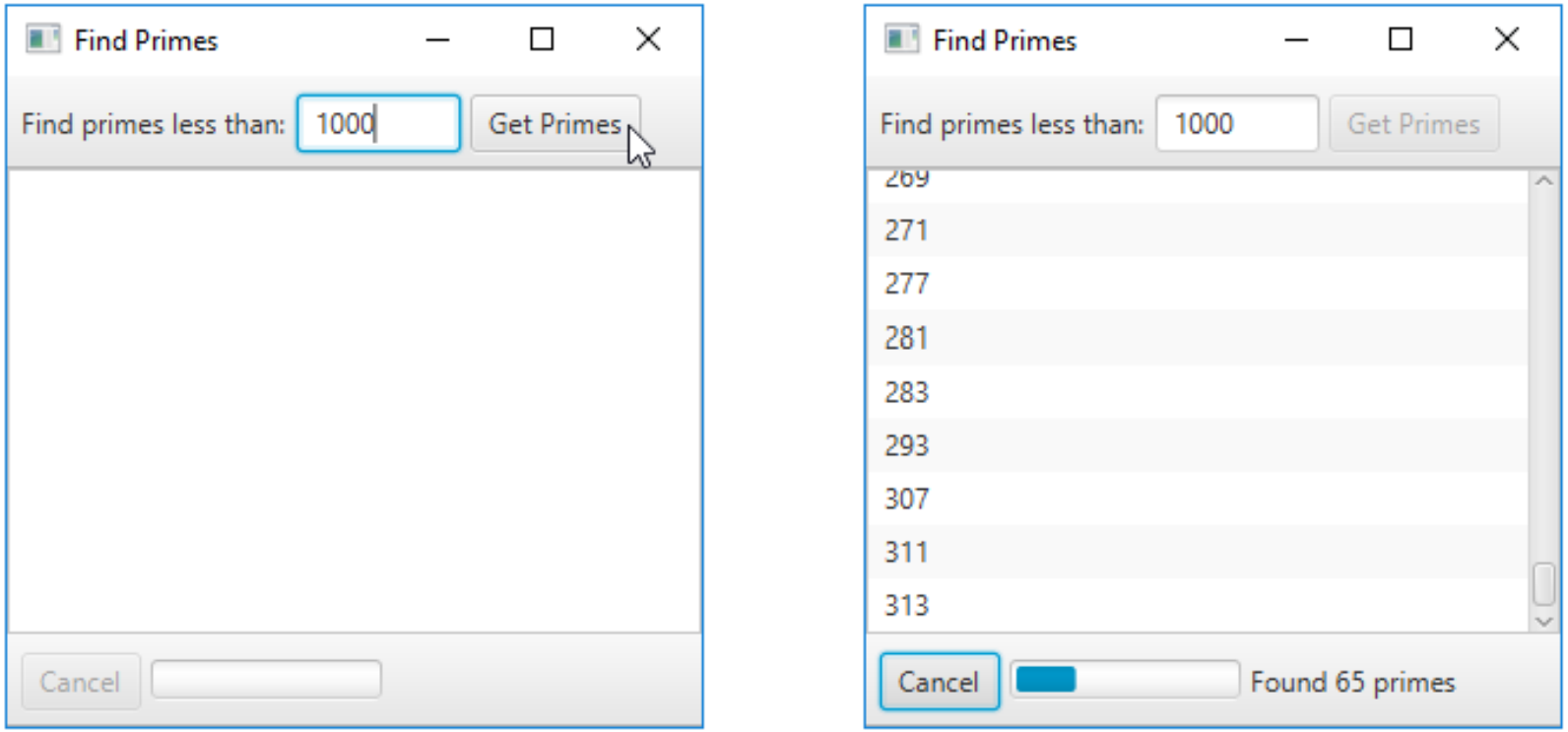


Fig. 23.28 | Displaying prime numbers as they're calculated and updating a `ProgressBar`. (Part 6 of 7.)

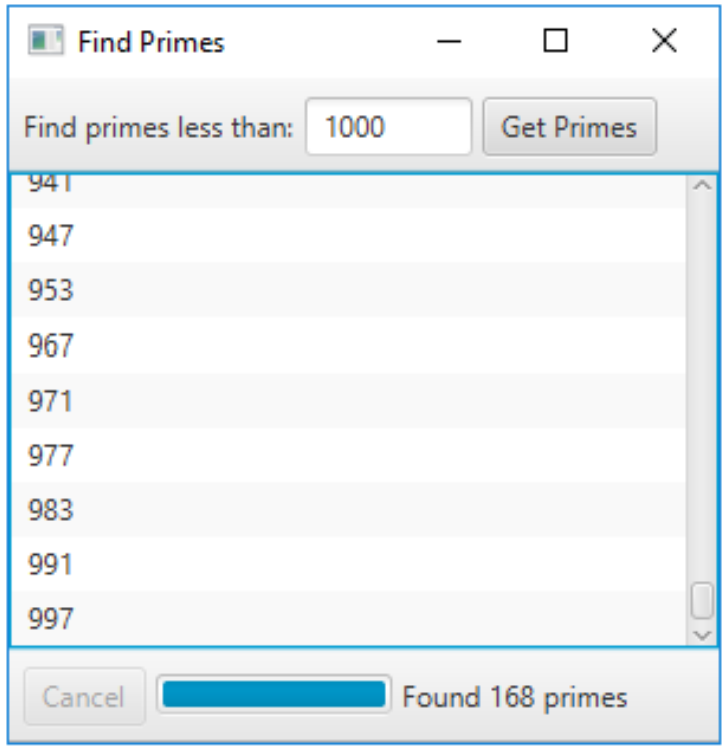


Fig. 23.28 | Displaying prime numbers as they're calculated and updating a `ProgressBar`. (Part 7 of 7.)

23.11.2 Processing Intermediate Results: Sieve of Eratosthenes (cont.)

Method getPrimesButtonPressed

- Creates a `PrimeCalculatorTask` then configures various property bindings and event listeners
- Binds the `statusLabel`'s `text` property to the task's message property to update the `statusLabel` automatically as new primes are found.
- Binds the `progressBar`'s `progress` property to the task's `progress` property to update the `progressBar` automatically with the percentage completion.

23.11.2 Processing Intermediate Results: Sieve of Eratosthenes (cont.)

Method getPrimesButtonPressed

- Registers a `ChangeListener` that gets invoked each time the task's value property changes
- Registers listeners for the task's *running* and *succeeded* state changes
  - Used to enable and disable the app's Buttons based on the task's state.
- Launches the task in a separate thread.

23.11.2 Processing Intermediate Results: Sieve of Eratosthenes (cont.)

Method cancelButtonPressed

- Calls the `PrimeCalculatorTask`'s inherited **`cancel`** method to terminate the task, then enables the `getPrimesButton` and disables the `cancelButton`.

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 quad-core system).

```
1  // Fig. 23.29: SortComparison.java
2  // Comparing performance of Arrays methods sort and parallelSort.
3  import java.time.Duration;
4  import java.time.Instant;
5  import java.text.NumberFormat;
6  import java.util.Arrays;
7  import java.util.Random;
8
9  public class SortComparison {
10    public static void main(String[] args) {
11       Random random = new Random();
12
13       // create array of random ints, then copy it
14       int[] array1 = random.ints(100_000_000).toArray();
15       int[] array2 = array1.clone();
16
17       // time the sorting of array1 with Arrays method sort
18       System.out.println("Starting sort");
19       Instant sortStart = Instant.now();
20       Arrays.sort(array1);
21       Instant sortEnd = Instant.now();
```

Fig. 23.29 | Comparing performance of `Arrays` methods `sort` and `parallelSort`. (Part 1 of 4.)

```
22
23          // display timing results
24          long sortTime = Duration.between(sortStart, sortEnd).toMillis();
25          System.out.printf("Total time in milliseconds: %d%n%n", sortTime);
26
27          // time the sorting of array2 with Arrays method parallelSort
28          System.out.println("Starting parallelSort");
29          Instant parallelSortStart = Instant.now();
30          Arrays.parallelSort(array2);
31          Instant parallelSortEnd = Instant.now();
32
```

Fig. 23.29 | Comparing performance of `Arrays` methods `sort` and `parallelSort`. (Part 2 of 4.)

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

Fig. 23.29 | Comparing performance of `Arrays` methods `sort` and `parallelSort`. (Part 3 of 4.)

```
Starting sort
Total time in milliseconds: 8883

Starting parallelSort
Total time in milliseconds: 2143

sort took 315% more time than parallelSort
```

Fig. 23.29 | Comparing performance of `Arrays` methods `sort` and `parallelSort`. (Part 4 of 4.)

23.12 `sort/parallelSort` Timings with the Java SE 8 Date/Time API (Cont.)

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`.

23.12 `sort/parallelSort` Timings with the Java SE 8 Date/Time API (Cont.)

- `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 to compare the performance.

```
1  // Fig. 23.30: 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.util.Random;
9
10 public class StreamStatisticsComparison {
11    public static void main(String[] args) {
12       Random random = new Random();
13
14       // create array of random long values
15       long[] values = random.longs(50_000_000, 1, 1001).toArray();
16
```

Fig. 23.30 | Comparing performance of sequential and parallel stream operations. (Part 1 of 5.)

```
17          // perform calculations separately
18          Instant separateStart = Instant.now();
19          long count = Arrays.stream(values).count();
20          long sum = Arrays.stream(values).sum();
21          long min = Arrays.stream(values).min().getAsLong();
22          long max = Arrays.stream(values).max().getAsLong();
23          double average = Arrays.stream(values).average().getAsDouble();
24          Instant separateEnd = Instant.now();
25
26          // display results
27          System.out.println("Calculations performed separately");
28          System.out.printf("    count: %,d%n", count);
29          System.out.printf("      sum: %,d%n", sum);
30          System.out.printf("      min: %,d%n", min);
31          System.out.printf("      max: %,d%n", max);
32          System.out.printf("  average: %f%n", average);
33          System.out.printf("Total time in milliseconds: %d%n%n",
34             Duration.between(separateStart, separateEnd).toMillis());
```

Fig. 23.30 | Comparing performance of sequential and parallel stream operations. (Part 2 of 5.)

```
35
36          // time summaryStatistics operation with sequential stream
37          LongStream stream1 = Arrays.stream(values);
38          System.out.println("Calculating statistics on sequential stream");
39          Instant sequentialStart = Instant.now();
40          LongSummaryStatistics results1 = stream1.summaryStatistics();
41          Instant sequentialEnd = Instant.now();
42
43          // display results
44          displayStatistics(results1);
45          System.out.printf("Total time in milliseconds: %d%n%n",
46             Duration.between(sequentialStart, sequentialEnd).toMillis());
47
48          // time sum operation with parallel stream
49          LongStream stream2 = Arrays.stream(values).parallel();
50          System.out.println("Calculating statistics on parallel stream");
51          Instant parallelStart = Instant.now();
52          LongSummaryStatistics results2 = stream2.summaryStatistics();
53          Instant parallelEnd = Instant.now();
```

Fig. 23.30 | Comparing performance of sequential and parallel stream operations. (Part 3 of 5.)

```
54
55          // display results
56          displayStatistics(results1);
57          System.out.printf("Total time in milliseconds: %d%n%n",
58             Duration.between(parallelStart, parallelEnd).toMillis());
59       }
60
61       // display's LongSummaryStatistics values
62       private static void displayStatistics(LongSummaryStatistics stats) {
63          System.out.println("Statistics");
64          System.out.printf("    count: %,d%n", stats.getCount());
65          System.out.printf("      sum: %,d%n", stats.getSum());
66          System.out.printf("      min: %,d%n", stats.getMin());
67          System.out.printf("      max: %,d%n", stats.getMax());
68          System.out.printf("  average: %f%n", stats.getAverage());
69       }
70    }
```

Fig. 23.30 | Comparing performance of sequential and parallel stream operations. (Part 4 of 5.)

```
Calculations performed separately
    count: 50,000,000
      sum: 25,025,212,218
      min: 1
      max: 1,000
  average: 500.504244
Total time in milliseconds: 710

Calculating statistics on sequential stream
Statistics
    count: 50,000,000
      sum: 25,025,212,218
      min: 1
      max: 1,000
  average: 500.504244
Total time in milliseconds: 305

Calculating statistics on parallel stream
Statistics
    count: 50,000,000
      sum: 25,025,212,218
      min: 1
      max: 1,000
  average: 500.504244
Total time in milliseconds: 143
```

Fig. 23.30 | Comparing performance of sequential and parallel stream operations. (Part 5 of 5.)

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

- The Callable interface (of package `java.util.concurrent`) declares a single method named call.

```
1  // Fig. 23.31: 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
9  // class that stores two Instants in time
10 class TimeData {
11    public Instant start;
12    public Instant end;
13
14    // return total time in seconds
15    public double timeInSeconds() {
16       return Duration.between(start, end).toMillis() / 1000.0;
17    }
18 }
19
```

Fig. 23.31 | Fibonacci calculations performed synchronously and asynchronously. (Part 1 of 7.)

```
20  public class FibonacciDemo {
21     public static void main(String[] args)
22        throws InterruptedException, ExecutionException {
23
24        // perform synchronous fibonacci(45) and fibonacci(44) calculations
25        System.out.println("Synchronous Long Running Calculations");
26        TimeData synchronousResult1 = startFibonacci(45);
27        TimeData synchronousResult2 = startFibonacci(44);
28        double synchronousTime =
29           calculateTime(synchronousResult1, synchronousResult2);
30        System.out.printf(
31           "  Total calculation time = %.3f seconds%n", synchronousTime);
32
33        // perform asynchronous fibonacci(45) and fibonacci(44) calculations
34        System.out.printf("%nAsynchronous Long Running Calculations%n");
35        CompletableFuture<TimeData> futureResult1 =
36           CompletableFuture.supplyAsync(() -> startFibonacci(45));
37        CompletableFuture<TimeData> futureResult2 =
38           CompletableFuture.supplyAsync(() -> startFibonacci(44));
```

Fig. 23.31 | Fibonacci calculations performed synchronously and asynchronously. (Part 2 of 7.)

```
39
40          // wait for results from the asynchronous operations
41          TimeData asynchronousResult1 = futureResult1.get();
42          TimeData asynchronousResult2 = futureResult2.get();
43          double asynchronousTime =
44             calculateTime(asynchronousResult1, asynchronousResult2);
45          System.out.printf(
46             "  Total calculation time = %.3f seconds%n", asynchronousTime);
47
48          // display time difference as a percentage
49          String percentage = NumberFormat.getPercentInstance().format(
50             (synchronousTime - asynchronousTime) / asynchronousTime);
51          System.out.printf("%nSynchronous calculations took %s" +
52             " more time than the asynchronous ones%n", percentage);
53       }
54
```

Fig. 23.31 | Fibonacci calculations performed synchronously and asynchronously. (Part 3 of 7.)

```
55      // executes function fibonacci asynchronously
56      private static TimeData startFibonacci(int n) {
57         // create a TimeData object to store times
58         TimeData timeData = new TimeData();
59
60         System.out.printf("  Calculating fibonacci(%d)%n", n);
61         timeData.start = Instant.now();
62         long fibonacciValue = fibonacci(n);
63         timeData.end = Instant.now();
64         displayResult(n, fibonacciValue, timeData);
65         return timeData;
66      }
67
68      // recursive method fibonacci; calculates nth Fibonacci number
69      private static long fibonacci(long n) {
70         if (n == 0 || n == 1) {
71            return n;
72         }
73         else {
74            return fibonacci(n - 1) + fibonacci(n - 2);
75         }
76      }
```

Fig. 23.31 | Fibonacci calculations performed synchronously and asynchronously. (Part 4 of 7.)

```
77
78      // display fibonacci calculation result and total calculation time
79      private static void displayResult(
80         int n, long value, TimeData timeData) {
81
82         System.out.printf("  fibonacci(%d) = %d%n", n, value);
83         System.out.printf(
84            "  Calculation time for fibonacci(%d) = %.3f seconds%n",
85            n, timeData.timeInSeconds());
86      }
87
```

Fig. 23.31 | Fibonacci calculations performed synchronously and asynchronously. (Part 5 of 7.)

```
88      // display fibonacci calculation result and total calculation time
89      private static double calculateTime(
90         TimeData result1, TimeData result2) {
91
92         TimeData bothThreads = new TimeData();
93
94         // determine earlier start time
95         bothThreads.start = result1.start.compareTo(result2.start) < 0 ?
96            result1.start : result2.start;
97
98         // determine later end time
99         bothThreads.end = result1.end.compareTo(result2.end) > 0 ?
100           result1.end : result2.end;
101
102        return bothThreads.timeInSeconds();
103     }
104  }
```

Fig. 23.31 | Fibonacci calculations performed synchronously and asynchronously. (Part 6 of 7.)

```
Synchronous Long Running Calculations
  Calculating fibonacci(45)
  fibonacci(45) = 1134903170
  Calculation time for fibonacci(45) = 4.395 seconds
  Calculating fibonacci(44)
  fibonacci(44) = 701408733
  Calculation time for fibonacci(44) = 2.722 seconds
  Total calculation time = 7.122 seconds

Asynchronous Long Running Calculations
  Calculating fibonacci(45)
  Calculating fibonacci(44)
  fibonacci(44) = 701408733
  Calculation time for fibonacci(44) = 2.707 seconds
  fibonacci(45) = 1134903170
  Calculation time for fibonacci(45) = 4.403 seconds
  Total calculation time = 4.403 seconds

Synchronous calculations took 62% more time than the asynchronous ones
```

Fig. 23.31 | Fibonacci calculations performed synchronously and asynchronously. (Part 7 of 7.)

23.15 (Advanced) Fork/Join Framework

- Java's concurrency APIs include the fork/join framework, which helps programmers parallelize algorithms.
- The fork/join framework is well suited to divide-and-conquer-style algorithms, such as the merge sort in Section 19.3.3.
- 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 parallel tasks to different processors are handled for you by the framework.