Chapter 23
Concurrency

Java How to Program, 11/e, Global Edition

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.

23.4 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`

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.

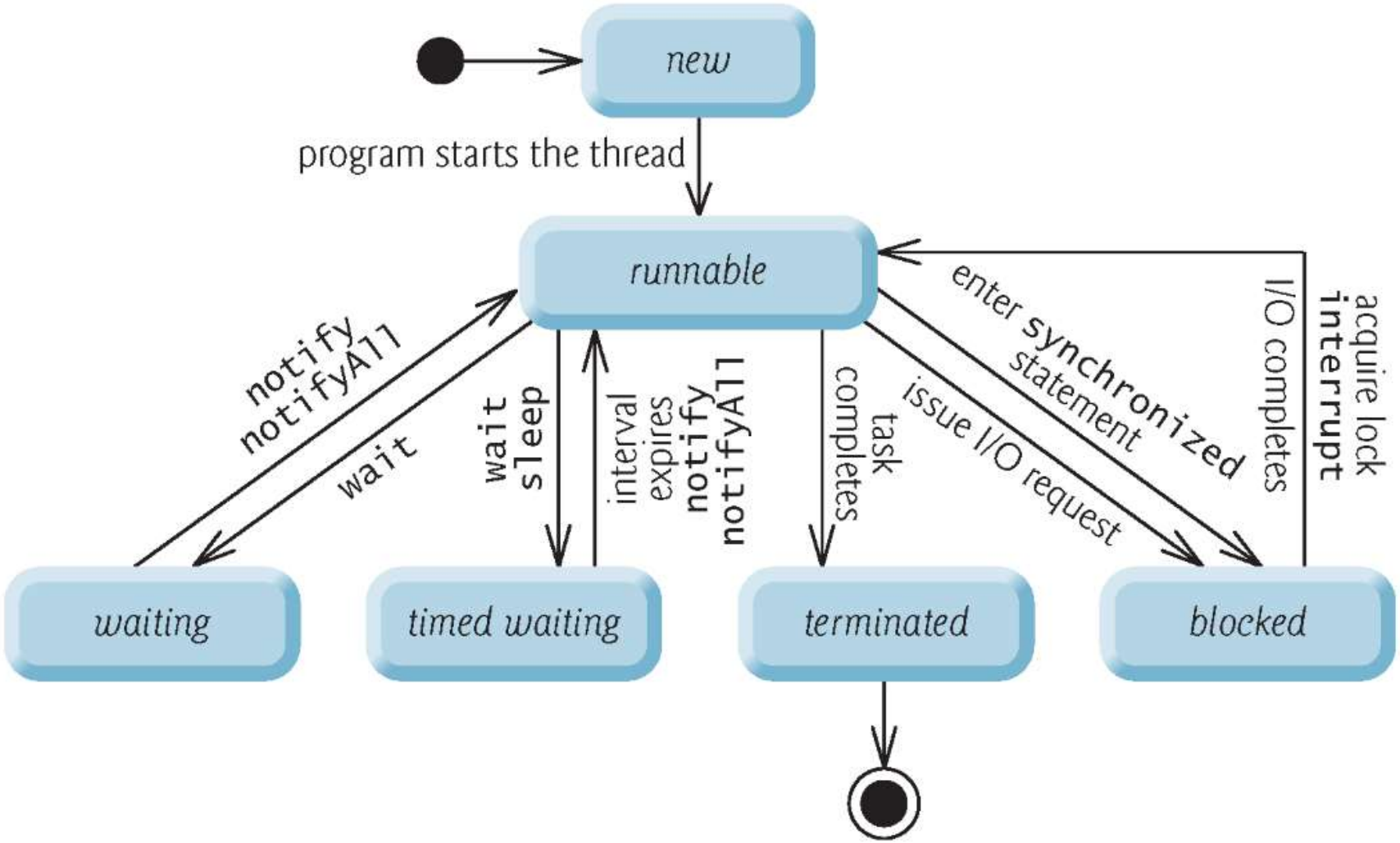

Fig. 23.1 | Thread life-cycle UML state diagram.

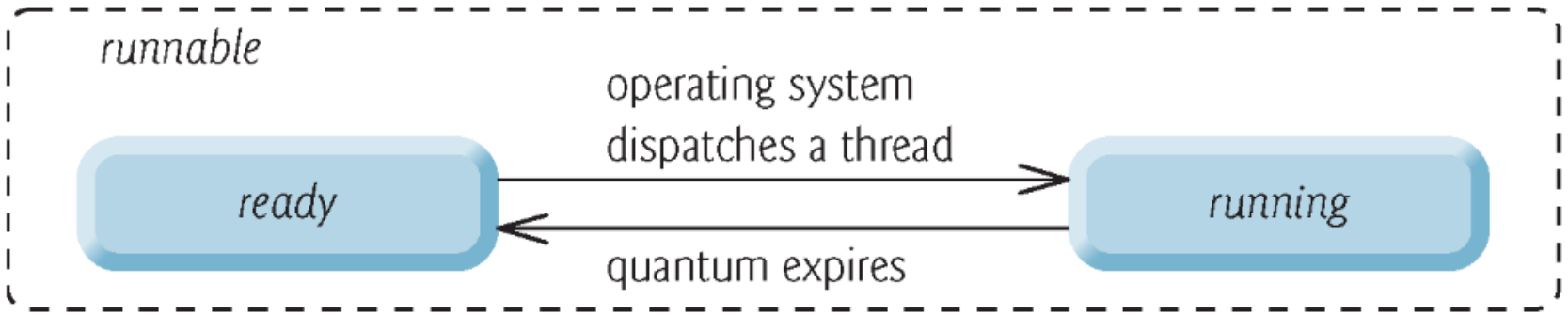

Fig. 23.2 | Operating system's internal view of Java's *runnable* state.

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.

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.

```
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.)

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.

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).

```
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.)

```
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.)

```
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.)

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.

```
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.

```
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.

```
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.)

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.)

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.)

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.

```
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.)

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.

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`.

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.

```
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.)

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`).

```
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.)

```
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.)

```
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.)