23.4 Thread Synchronization

When multiple threads share an object and it's *modified* by one or more of them, indeterminate results may occur (as we'll see in the examples) unless access to the shared object is managed properly. If one thread is in the process of updating a shared object and another thread also tries to update it, it's uncertain which thread's update takes effect. Similarly, if one thread is in the process of updating a shared object and another thread tries to read it, it's uncertain whether the reading thread will see the old value or the new one. In such cases, the program's behavior cannot be trusted—sometimes the program will produce the correct results, and sometimes it won't, and there won't be any indication that the shared object was manipulated incorrectly.

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

mutual exclusion.

23.4.1 Immutable Data

Actually, thread synchronization is necessary *only* for shared **mutable data**, i.e., data that may *change* during its lifetime. With shared **immutable data** that will *not* change, it's not possible for a thread to see old or incorrect values as a result of another thread's manipulation of that data.

When you share *immutable data* across threads, declare the corresponding data fields `final` to indicate that the variables's values will *not* change after they're initialized. This prevents accidental modification of the shared data, which could compromise thread safety. *Labeling object references as* `final` *indicates that the reference will not change, but it does not guarantee that the referenced object is immutable—this depends entirely on the object's properties.* But, it's still good practice to mark references that will not change as `final`.

 Software Engineering Observation 23.3

Always declare data fields that you do not expect to change as `final`. *Primitive variables that are declared as* `final` *can safely be shared across threads. An object reference that's declared as* `final` *ensures that the object it refers to will be*

fully constructed and initialized before it's used by the program, and prevents the reference from pointing to another object.

23.4.2 Monitors

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

To specify that a thread must hold a monitor lock to execute a block of code, the code should be placed in a `synchronized` **statement**. Such code is said to be **guarded** by the monitor lock; a thread must **acquire the lock** to execute the guarded statements. The monitor allows only one thread at a time to execute statements within `synchronized` statements that lock on the same object, as only one thread at a time can hold the monitor lock. The `synchronized` statements are declared using the `synchronized` **keyword**:

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

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

When a `synchronized` statement finishes executing, the object's monitor lock is released and one of the *blocked* threads attempting to enter a `synchronized` statement can be allowed to acquire the lock to proceed. Java also allows `synchronized` **methods**. Before executing, a `synchronized` instance method must acquire the lock on the object that's used to call the method. Similarly, a `static synchronized` method must acquire the lock on a `Class` object that represents the class in which the method is declared. A `Class` object is the execution-time representation of a class that the JVM has loaded into memory.

Software Engineering

Observation 23.4

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

23.4.3 Unsynchronized Mutable Data Sharing

First, we illustrate the dangers of sharing an object across threads *without* proper synchronization. In this example (Figs. 23.5–23.7), two `Runnable`s maintain references to a single integer array. Each `Runnable` writes three values to the array, then terminates. This may seem harmless, but we'll see that it can result in errors if the array is manipulated without synchronization.

Class `SimpleArray`

We'll *share* a `SimpleArray` object (Fig. 23.5) across multiple threads. `SimpleArray` will enable those threads to place `int` values into `array` (declared at line 8). Line 9 initializes variable `writeIndex`, which determines the array element that should be written to next. The constructor (line 12) creates an integer array of the desired size.

```
1  // Fig. 23.5: SimpleArray.java
2  // Class that manages an integer array to be sha
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5
6  public class SimpleArray { // CAUTION: NOT THREA
7     private static final SecureRandom generator =
8     private final int[] array; // the shared inte
9     private int writeIndex = 0; // shared index o
10
11    // construct a SimpleArray of a given size
12    public SimpleArray(int size) {array = new int
13
14    // add a value to the shared array
15    public void add(int value) {
16       int position = writeIndex; // store the wr
17
18       try {
19          // put thread to sleep for 0-499 millis
20          Thread.sleep(generator.nextInt(500));
21       }
22       catch (InterruptedException ex) {
23          Thread.currentThread().interrupt(); //
24       }
25
26       // put value in the appropriate element
27       array[position] = value;
28       System.out.printf("%s wrote %2d to element
29          Thread.currentThread().getName(), value
30
31       ++writeIndex; // increment index of elemen
32       System.out.printf("Next write index: %d%n"
33    }
34
35    // used for outputting the contents of the sh
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.)

Method `add` (lines 15–33) places a new value into the array. Line 16 stores the current `writeIndex` value. Line 20 puts the thread that invokes `add` to sleep for a random interval from 0 to 499 milliseconds. We do this for demonstration purposes to make the problems associated with *unsynchronized access to shared mutable data* more obvious. After the thread is done sleeping, line 27 inserts the value passed to `add` into the array at the element specified by `position`. Lines 28–29 display the executing thread's name, the value that was added and the value's index in the array. In line 29, the expression

```
Thread.currentThread().getName()
```

first obtains a reference to the currently executing `Thread`, then uses its `getName` method to obtain its name. Line 31 increments `writeIndex` so that the next call to `add` will insert a value in the array's next element. Lines 36–39 override method `toString` to create a `String` representation of the array's contents.

Class `ArrayWriter`

Class `ArrayWriter` (Fig. 23.6) implements the interface `Runnable` to define a task for inserting values in a `SimpleArray` object. The constructor (lines 9–12) receives an `int` representing the first value this task will insert in a `SimpleArray` object and a reference to the `SimpleArray` object to manipulate. Line 17 invokes `SimpleArray` method `add`. The task completes after lines 16–18 `add` three consecutive integers beginning with `startValue`.

```
1   // Fig. 23.6: ArrayWriter.java
2   // Adds integers to an array shared with other R
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 arr
10        startValue = value;
11        sharedSimpleArray = array;
12     }
13
14     @Override
15     public void run() {
16        for (int i = startValue; i < startValue +
17           sharedSimpleArray.add(i); // add an ele
18        }
19     }
20  }
```

Fig. 23.6

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

Class `SharedArrayTest`

Class `SharedArrayTest` (Fig. 23.7) executes two `ArrayWriter` tasks that add values to a single `SimpleArray` object. Line 10 constructs a six-element `SimpleArray` object. Lines 13–14 create two new `ArrayWriter` tasks, one that places the values `1` through `3` in the `SimpleArray` object, and one that places the values `11` through `13`. Lines 17–19 create an `ExecutorService` and execute the two `ArrayWriter`s. Line 21 invokes the `ExecutorService`'s `shutDown` method to prevent it from accepting additional tasks and to enable the application to terminate when the currently executing tasks complete execution.

```
 1  // Fig. 23.7: SharedArrayTest.java
 2  // Executing two Runnables to add elements to a
 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 Simple
```

```
                    11
12          // create two tasks to write to the shared
13          ArrayWriter writer1 = new ArrayWriter(1, s
14          ArrayWriter writer2 = new ArrayWriter(11,
                    15
16          // execute the tasks with an ExecutorServi
17          ExecutorService executorService = Executor
18             executorService.execute(writer1);
19             executorService.execute(writer2);
                    20
21             executorService.shutdown();
                    22
               23          try {
24             // wait 1 minute for both writers to fi
25                boolean tasksEnded =
26                executorService.awaitTermination(1,
                    27
28                if (tasksEnded) {
29                System.out.printf("%nContents of Sim
30                System.out.println(sharedSimpleArray
                  31             }
                32             else {
33                System.out.println(
34                   "Timed out while waiting for task
                  35             }
                    36          }
37          catch (InterruptedException ex) {
38             ex.printStackTrace();
                    39          }
                      40       }
                        41   }
```

```
pool-1-thread-1 wrote  1 to element 0. — pool-1-threa
               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-threa
                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.)

`ExecutorService` Method `awaitTermination`

Recall that `ExecutorService` method `shutdown` returns immediately. Thus any code that appears *after* the call to `ExecutorService` method `shutdown` in line 21 *will continue executing as long as the* `main` *thread is still assigned to a processor*. We'd like to output the `SimpleArray` object to show you the results *after* the threads complete their tasks. So, we need the program to wait for the threads to complete before `main` outputs the `SimpleArray` object's contents.

Interface `ExecutorService` provides the `awaitTermination` method for this purpose. This method returns control to its caller either when all tasks executing in the `ExecutorService` complete or when the specified timeout elapses. If all tasks are completed before `awaitTermination` times out, this method returns `true`; otherwise it returns `false`. The two arguments to `awaitTermination` represent a timeout value and a unit of measure specified with a constant from class `TimeUnit` (in this case, `TimeUnit.MINUTES`).

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

In this example, if *both* tasks complete before `awaitTermination` times out, line 30 displays the `SimpleArray` object's contents. Otherwise, lines 33–34 display a message indicating that the tasks did not finish executing before `awaitTermination` timed out.

Sample Program Output

Figure 23.7's output shows the problems (highlighted in the output) that can be caused by *failure to synchronize access to shared mutable data*. The value `1` was written to element `0`,

then *overwritten* later by the value `11`. Also, when `writeIndex` was incremented to 3, *nothing was written to that element*, as indicated by the 0 in that element of the array.

Recall that we call `Thread` method `sleep` between operations on the shared mutable data to emphasize the *unpredictability of thread scheduling* and to increase the likelihood of producing erroneous output. Even if these operations were allowed to proceed at their normal pace, you could still see errors in the program's output. However, modern processors can handle `SimpleArray` method `add`'s operations so quickly that you might not see the errors caused by the two threads executing this method concurrently, even if you tested the program dozens of times.

One of the challenges of multithreaded programming is spotting the errors—they may occur so infrequently and unpredictably that a broken program does not produce incorrect results during testing, creating the illusion that the program is correct. This is all the more reason to use predefined collections that handle the synchronization for you.

23.4.4 Synchronized Mutable Data Sharing—Making Operations Atomic

Figure 23.7's output errors can be attributed to the fact that the shared `SimpleArray` is not **thread safe**—it's susceptible to

errors if it's *accessed concurrently by multiple threads*. The problem lies in method `add`, which stores the `writeIndex` value, places a new value in that element, then increments `writeIndex`. This would not present a problem in a single-threaded program. However, if one thread obtains the `writeIndex` value, there's no guarantee that another thread will not 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`—that is, a value that's no longer valid. Another possibility is that one thread might obtain the `writeIndex` value *after* another thread adds an element to the array but *before* `writeIndex` is incremented. In this case, too, the first thread would use an invalid `writeIndex` value.

`SimpleArray` is *not thread safe because it allows any number of threads to read and modify shared mutable data concurrently*, which can cause errors. To make `SimpleArray` thread safe, we must ensure that no two threads can access its shared mutable data at the same time. While one thread is in the process of storing `writeIndex`, adding a value to the array, and incrementing `writeIndex`, *no other thread* may read or change the value of `writeIndex` or modify the contents of the array at any point during these three operations. In other words, we want these three operations—storing `writeIndex`, writing to the array, incrementing `writeIndex`—to be an **atomic operation**, which cannot be divided into smaller suboperations. (As you'll see in later examples, read operations on shared mutable data

should also be atomic.) We can simulate atomicity by ensuring that only one thread carries out the three operations at a time. Any other threads that need to perform the operation must *wait* until the first thread has finished the `add` operation in its entirety.

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

Software Engineering Observation 23.5

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

Class `SimpleArray` with Synchronization

Figure 23.8 displays class `SimpleArray` with the proper synchronization. Notice that it's identical to the `SimpleArray` class of Fig. 23.5, except that `add` is now a `synchronized` method (line 18 of Fig. 23.8). So, only one thread at a time can execute this method. We reuse classes `ArrayWriter` (Fig. 23.6) and `SharedArrayTest` (Fig. 23.7) from the previous example, so we do not show them again here.

```
   1  // Fig. 23.8: SimpleArray.java
2  // Class that manages an integer array to be sha
   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 =
9     private final int[] array; // the shared inte
10    private int writeIndex = 0; // index of next
                  11
12    // construct a SimpleArray of a given size
13       public SimpleArray(int size) {
   14       array = new int[size];
               15    }
                  16
17    // add a value to the shared array
18    public synchronized void add(int value) {
19       int position = writeIndex; // store the wr
                  20
            21       try {
22          // in real applications, you shouldn't
```

```
23            Thread.sleep(generator.nextInt(500)); /
                  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
32             Thread.currentThread().getName(), value
                       33
34          ++writeIndex; // increment index of elemen
35          System.out.printf("Next write index: %d%n"
                    36      }
                       37
38       // used for outputting the contents of the sh
               39      @Override
 40       public synchronized String toString() {
    41          return Arrays.toString(array);
                  42      }
                    43   }
```

```
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.

Line 18 declares method `add` as `synchronized`, making all of the operations in this method behave as a single, atomic operation. Line 19 performs the first suboperation—storing the value of `writeIndex`. Line 30 performs the second suboperation, writing a value to the element at the index `position`. Line 34 performs the third suboperation, incrementing `writeIndex`. When the method finishes executing at line 36, the executing thread implicitly *releases* the `SimpleArray` object's lock, making it possible for another thread to begin executing the `add` method.

In the `synchronized add` method, we print messages to the console indicating the progress of threads as they execute this method, in addition to performing the actual operations required to insert a value in the array. We do this so that the messages will be printed in the correct order, allowing us to see whether the method is properly synchronized by comparing these outputs with those of the previous, unsynchronized example. We continue to output messages from `synchronized` blocks in later examples for *demonstration purposes only*; typically, however, I/O *should not* be performed in `synchronized` blocks, because it's important to minimize the amount of time that an object is "locked." **[*Note:* Line 23 in this example calls `Thread` method `sleep` (for demo purposes only) to emphasize the**

unpredictability of thread scheduling. You should never call `sleep` while holding a lock in a real application.]

Performance Tip 23.2

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