Chapter 23 Concurrency

Java How to Program, 10/e

OBJECTIVES

In this chapter you'll:

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

23.1 Introduction

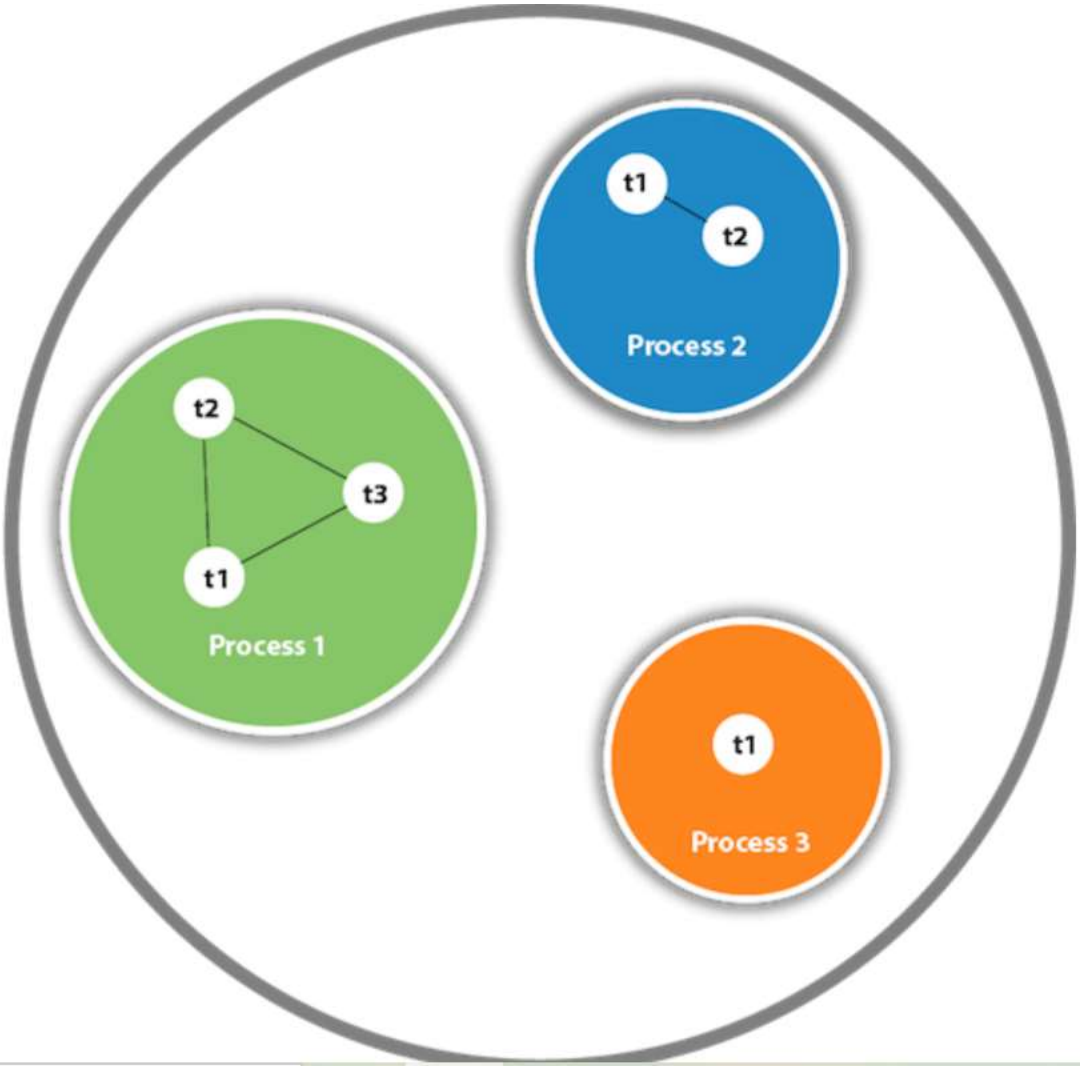
- When we say that two tasks are operating concurrently, we mean that they're both *making progress* at once.
- When we say that two tasks are operating in parallel, we mean that they're executing *simultaneously*.
- Java makes concurrency available to you through the language and APIs.
- You specify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

Performance Tip 23.1

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

23.1 Introduction (cont.)

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



23.2 Thread States and Life Cycle

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

23.2.1 *New* and *Runnable* States

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

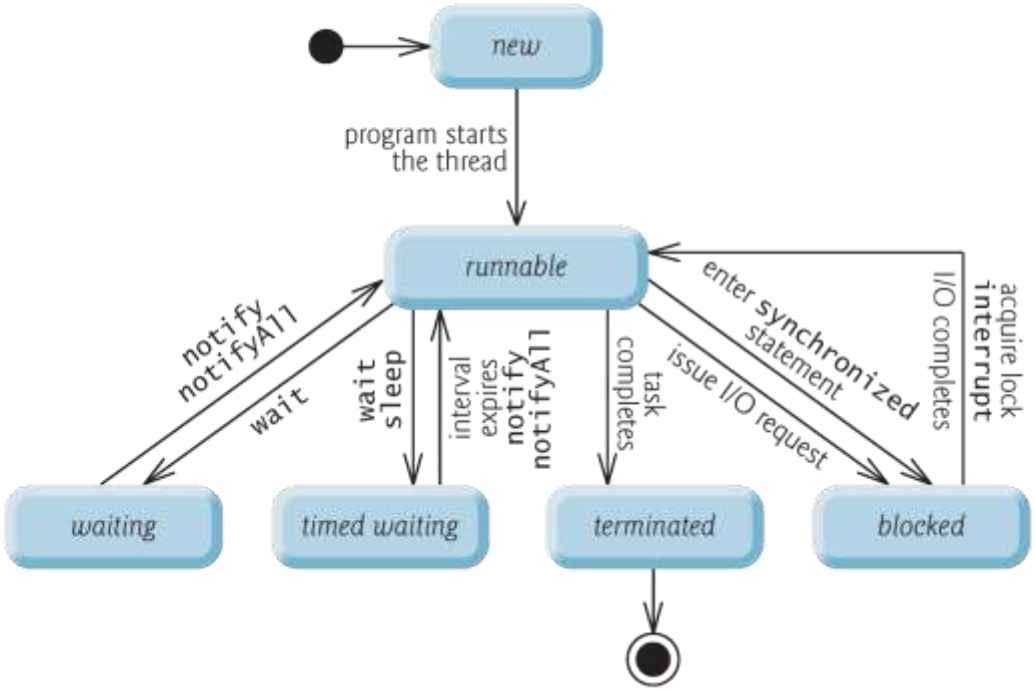


Fig. 23.1 | Thread life-cycle UML state diagram.

23.2.1 *New* and *Runnable* States

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

23.2.2 *Waiting* State

- A *runnable* thread can transition to the *waiting* state while it waits for another thread to perform a task.
  - Transitions back to the runnable state only when another thread notifies it to continue executing.

23.2.3 *Timed Waiting* State

- A *runnable* thread can enter the *timed waiting* state for a specified interval of time.
  - Transitions back to the *runnable* state when that time interval expires or when the event it's waiting for occurs.
  - Cannot use a processor, even if one is available.
- A sleeping thread remains in the *timed waiting* state for a designated period of time (called a sleep interval), after which it returns to the *runnable* state.

23.2.4 *Blocked* State

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

23.2.5 *Terminated* State

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

23.2.6 Operating-System View of the *Runnable* State

- At the operating-system level, Java's *runnable* state typically encompasses two separate states (Fig. 23.2).
- When a thread first transitions to the *runnable* state from the new state, it is in the *ready* state.
- A *ready* thread enters the *running* state (i.e., begins executing) when the operating system assigns it to a processor—also known as dispatching the thread.
- Typically, each thread is given a a quantum or timeslice in which to perform its task.
- The process that an operating system uses to determine which thread to dispatch is called thread scheduling.

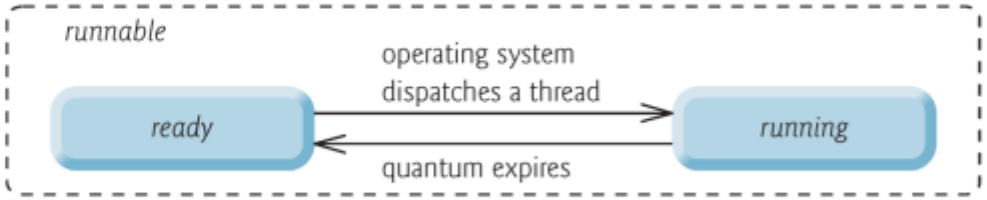


Fig. 23.2 | Operating system's internal view of Java's *runnable* state.

The Thread Class and the Runnable Interface

- Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable. Thread encapsulates a thread of execution.
- To create a new thread, your program will either extend Thread or implement the Runnable interface.

Java Thread class

- Java provides **Thread class** to achieve thread programming.
- Thread class provides constructors and methods to create and perform operations on a thread.
- Thread class extends Object class and implements Runnable interface.

Java Thread class

The **Thread** class defines several methods that help manage threads. Several of those used in this chapter are shown here:

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Java Thread class

- Although the main thread is created automatically when your program is started, it can be controlled through a Thread object.
- To do so, you must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread. Its general form is shown here:
- static Thread currentThread( )
- This method returns a reference to the thread in which it is called.

23.2.7 Thread Priorities and Thread Scheduling

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

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.

Creating Threads

- You can implement the Runnable interface.
- You can extend the Thread class, itself.

Implementing Runnable

The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this:

```
public void run( )
```

Inside **run( )**, you will define the code that constitutes the new thread. It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( )** returns.

- After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class.
- **Thread** defines several constructors. The one that we will use is shown here:
  - Thread(Runnable *threadOb*, String *threadName*)
- In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin.
- The name of the new thread is specified by *threadName*.
- After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**.
- In essence, **start( )** initiates a call to **run( )**. The **start( )** method is shown here:
  - void start( )

```
// Create a second thread.
class NewThread implements Runnable {
  Thread t;

  NewThread() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread");
    System.out.println("Child thread: " + t);
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}
```

```
class ThreadDemo {
  public static void main(String args[]) {
    NewThread nt = new NewThread(); // create a new thread

    nt.t.start(); // Start the thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

Extending Thread

- The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class.
- The extending class must override the run( ) method, which is the entry point for the new thread.
- As before, a call to start( ) begins execution of the new thread.

```
// Create a second thread by extending Thread
class NewThread extends Thread {

  NewThread() {
    // Create a new, second thread
    super("Demo Thread");
    System.out.println("Child thread: " + this);
  }

  // This is the entry point for the second thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Child Thread: " + i);
        Thread.sleep(500);
      }
    } catch (InterruptedException e) {
      System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
  }
}
```

```
class ExtendThread {
  public static void main(String args[]) {
    NewThread nt = new NewThread(); // create a new thread

    nt.start(); // start the thread

    try {
      for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }
    System.out.println("Main thread exiting.");
  }
}
```

Choosing an Approach

- At this point, you might be wondering why Java has two ways to create child threads, and which approach is better.
- The **Thread** class defines several methods that can be overridden by a derived class. Of these methods, the only one that *must* be overridden is **run( )**.
- This is, of course, the same method required when you implement **Runnable**.
- Many Java programmers feel that classes should be extended only when they are being enhanced or adapted in some way.
- So, if you will not be overriding any of **Thread**'s other methods, it is probably best simply to implement **Runnable**.
- Also, by implementing **Runnable**, your thread class does not need to inherit **Thread**, making it free to inherit a different class.

Using isAlive( ) and join( )

final boolean isAlive( )

- The isAlive( ) method returns true if the thread upon which it is called is still running. It returns false otherwise.
- While isAlive( ) is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called join( ), shown here:
  - final void join( ) throws InterruptedException
- This method waits until the thread on which it is called terminates.

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("New thread: " + t);
  }

  // This is the entry point for thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
      }
    } catch (InterruptedException e) {
      System.out.println(name + " interrupted.");
    }
    System.out.println(name + " exiting.");
  }
}
```

```
    try {
      System.out.println("Waiting for threads to finish.");
      nt1.t.join();
      nt2.t.join();
      nt3.t.join();
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

    System.out.println("Thread One is alive: "
                       + nt1.t.isAlive());
    System.out.println("Thread Two is alive: "
                       + nt2.t.isAlive());
    System.out.println("Thread Three is alive: "
                       + nt3.t.isAlive());

    System.out.println("Main thread exiting.");
  }
```

23.3 Creating and Executing Threads with the Executor Framework

- A Runnable object represents a "task" that can execute concurrently with other tasks.
- The `Runnable` interface declares the single method run, which contains the code that defines the task that a `Runnable` object should perform.
- When a thread executing a `Runnable` is created and started, the thread calls the `Runnable` object's `run` method, which executes in the new thread.

Software Engineering Observation 23.2

Though it's possible to create threads explicitly, it's recommended that you use the `Executor` *interface to manage the execution of* `Runnable` *objects.*

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

- Class `PrintTask` (Fig. 23.3) implements `Runnable` (line 5), so that multiple `PrintTasks` can execute concurrently.
- `Thread` `static` method sleep places a thread in the *timed waiting state for the specified amount of time.*
  - Can throw a checked exception of type InterruptedException if the sleeping thread's interrupt method is called.
- The code in `main` executes in the main thread, a thread created by the JVM.
- The code in the `run` method of `PrintTask` executes in the threads created in `main`.
- When method `main` terminates, the program itself continues running because there are still threads that are alive.
  - The program will not terminate until its last thread completes execution.

```
1  // Fig. 23.3: PrintTask.java
2  // PrintTask class sleeps for a random time from 0 to 5 seconds
3  import java.security.SecureRandom;
4
5  public class PrintTask implements Runnable
6  {
7     private static final SecureRandom generator = new SecureRandom();
8     private final int sleepTime; // random sleep time for thread
9     private final String taskName;
10
11    // constructor
12    public PrintTask(String taskName)
13    {
14       this.taskName = taskName;
15
16       // pick random sleep time between 0 and 5 seconds
17       sleepTime = generator.nextInt(5000); // milliseconds
18    }
19
```

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

```
20      // method run contains the code that a thread will execute
21      public void run()
22      {
23         try // put thread to sleep for sleepTime amount of time
24         {
25            System.out.printf("%s going to sleep for %d milliseconds.%n",
26               taskName, sleepTime);
27            Thread.sleep(sleepTime); // put thread to sleep
28         }
29         catch (InterruptedException exception)
30         {
31            exception.printStackTrace();
32            Thread.currentThread().interrupt(); // re-interrupt the thread
33         }
34
35         // print task name
36         System.out.printf("%s done sleeping%n", taskName);
37      }
38   } // end class PrintTask
```

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

```
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  {
8     public static void main(String[] args)
9     {
10       // create and name each runnable
11       PrintTask task1 = new PrintTask("task1");
12       PrintTask task2 = new PrintTask("task2");
13       PrintTask task3 = new PrintTask("task3");
14
15       System.out.println("Starting Executor");
16
17       // create ExecutorService to manage threads
18       ExecutorService executorService = Executors.newCachedThreadPool();
19
20       // start the three PrintTasks
21       executorService.execute(task1); // start task1
22       executorService.execute(task2); // start task2
23       executorService.execute(task3); // start task3
24
```

Fig. 23.4 | Using an `ExecutorService` to execute `Runnable`s. (Part 1 of 3.)

```
25          // shut down ExecutorService--it decides when to shut down threads
26          executorService.shutdown();
27
28          System.out.printf("Tasks started, main ends.%n%n");
29       }
30    } // end class TaskExecutor
```

```
Starting Executor
Tasks started, main ends

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

Fig. 23.4 | Using an `ExecutorService` to execute `Runnable`s. (Part 2 of 3.)

```
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 3 of 3.)

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

- Recommended that you use the Executor interface to manage the execution of `Runnable` objects for you.
  - Typically creates and manages a group of threads called a thread pool to execute `Runnable`s.
- `Executor`s can reuse existing threads and can improve performance by optimizing the number of threads.
- `Executor` method execute accepts a `Runnable` as an argument.
- An `Executor` assigns every `Runnable` passed to its `execute` method to one of the available threads in the thread pool.
- If there are no available threads, the `Executor` creates a new thread or waits for a thread to become available.

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

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

23.4 Thread Synchronization

- When multiple threads share an object and it is *modified* by one or more of them, indeterminate results may occur unless access to the shared object is managed properly.
- The problem can be solved by giving only one thread at a time *exclusive access* to code that accesses the shared object.
  - During that time, other threads desiring to access the object are kept waiting.
  - When the thread with exclusive access finishes accessing the object, one of the waiting threads is allowed to proceed.
- This process, called thread synchronization, coordinates access to shared data by multiple concurrent threads.
  - Ensures that each thread accessing a shared object *excludes* all other threads from doing so simultaneously—this is called mutual exclusion.

23.4.1 Immutable Data

- Thread synchronization is necessary *only* for shared mutable data, i.e., data that may *change* during its lifetime.
- With shared immutable data that will *not* change, it's not possible for a thread to see old or incorrect values as a result of another thread's manipulation of that data.
- When you share *immutable data* across threads, declare the corresponding data fields `final` to indicate that the values of the variables will *not* change after they're initialized.

23.4.2 Monitors

- A common way to perform synchronization is to use Java's built-in monitors.
  - Every object has a monitor and a monitor lock (or intrinsic lock).
  - Can be held by a maximum of only one thread at any time.
  - A thread must acquire the lock before proceeding with the operation.
  - Other threads attempting to perform an operation that requires the same lock will be *blocked*.
- To specify that a thread must hold a monitor lock to execute a block of code, the code should be placed in a synchronized statement.
  - Said to be guarded by the monitor lock

23.4.2 Monitors (Cont.)

- The `synchronized` statements are declared using the synchronized keyword:
  - ```
    synchronized (object)
    {
        statements
    } // end synchronized statement
    ```
- where *object* is the object whose monitor lock will be acquired
  - *object* is normally `this` if it's the object in which the `synchronized` statement appears.
- When a `synchronized` statement finishes executing, the object's monitor lock is released.
- Java also allows synchronized methods.
    ```

Software Engineering Observation 23.4

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

23.4.3 Unsynchronized Mutable Data Sharing

- A `SimpleArray` object (Fig. 23.5) will be *shared* across multiple threads.
- Will enable those threads to place `int` values into `array`.
- Line 26 puts the thread that invokes `add` to sleep for a random interval from 0 to 499 milliseconds.
  - This is done to make the problems associated with *unsynchronized access to shared mutable data* more obvious.

```
1  // Fig. 23.5: SimpleArray.java
2  // Class that manages an integer array to be shared by multiple threads.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5
6  public class SimpleArray // CAUTION: NOT THREAD SAFE!
7  {
8     private static final SecureRandom generator = new SecureRandom();
9     private final int[] array; // the shared integer array
10    private int writeIndex = 0; // shared index of next element to write
11
12    // construct a SimpleArray of a given size
13    public SimpleArray(int size)
14    {
15       array = new int[size];
16    }
17
18    // add a value to the shared array
19    public void add(int value)
20    {
21       int position = writeIndex; // store the write index
22
```

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.)

```
23          try
24          {
25             // put thread to sleep for 0-499 milliseconds
26             Thread.sleep(generator.nextInt(500));
27          }
28          catch (InterruptedException ex)
29          {
30             Thread.currentThread().interrupt(); // re-interrupt the thread
31          }
32
33          // put value in the appropriate element
34          array[position] = value;
35          System.out.printf("%s wrote %2d to element %d.%n",
36             Thread.currentThread().getName(), value, position);
37
38          ++writeIndex; // increment index of element to be written next
39          System.out.printf("Next write index: %d%n", writeIndex);
40       }
41
```

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.)

```
42      // used for outputting the contents of the shared integer array
43      public String toString()
44      {
45         return Arrays.toString(array);
46      }
47   } // end class SimpleArray
```

Fig. 23.5 | Class that manages an integer array to be shared by multiple threads. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 3 of 3.)

23.4.3 Unsynchronized Data Sharing (cont.)

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

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

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

23.4.3 Unsynchronized Data Sharing (cont.)

- Class `SharedArrayTest` (Fig. 23.7) executes two `ArrayWriter` tasks that add values to a single `SimpleArray` object.
- `ExecutorService`'s `shutDown` method prevents additional tasks from starting and to enable the application to terminate when the currently executing tasks complete execution.
- We'd like to output the `SimpleArray` object to show you the results *after* the threads complete their tasks.
  - So, we need the program to wait for the threads to complete before `main` outputs the `SimpleArray` object's contents.
  - Interface `ExecutorService` provides the awaitTermination method for this purpose—returns control to its caller either when all tasks executing in the `ExecutorService` complete or when the specified timeout elapses.

```
1  // Fig. 23.7: SharedArrayTest.java
2  // Executing two Runnables to add elements to a shared SimpleArray.
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.ExecutorService;
5  import java.util.concurrent.TimeUnit;
6
7  public class SharedArrayTest
8  {
9     public static void main(String[] arg)
10    {
11       // construct the shared object
12       SimpleArray sharedSimpleArray = new SimpleArray(6);
13
14       // create two tasks to write to the shared SimpleArray
15       ArrayWriter writer1 = new ArrayWriter(1, sharedSimpleArray);
16       ArrayWriter writer2 = new ArrayWriter(11, sharedSimpleArray);
17
18       // execute the tasks with an ExecutorService
19       ExecutorService executorService = Executors.newCachedThreadPool();
20       executorService.execute(writer1);
21       executorService.execute(writer2);
22
23       executorService.shutdown();
```

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

```
24
25          try
26          {
27             // wait 1 minute for both writers to finish executing
28             boolean tasksEnded =
29                executorService.awaitTermination(1, TimeUnit.MINUTES);
30
31             if (tasksEnded)
32             {
33                System.out.printf("%nContents of SimpleArray:%n");
34                System.out.println(sharedSimpleArray); // print contents
35             }
36             else
37                System.out.println(
38                   "Timed out while waiting for tasks to finish.");
39          }
40          catch (InterruptedException ex)
41          {
42             ex.printStackTrace();
43          }
44       } // end main
45    } // end class SharedArrayTest
```

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

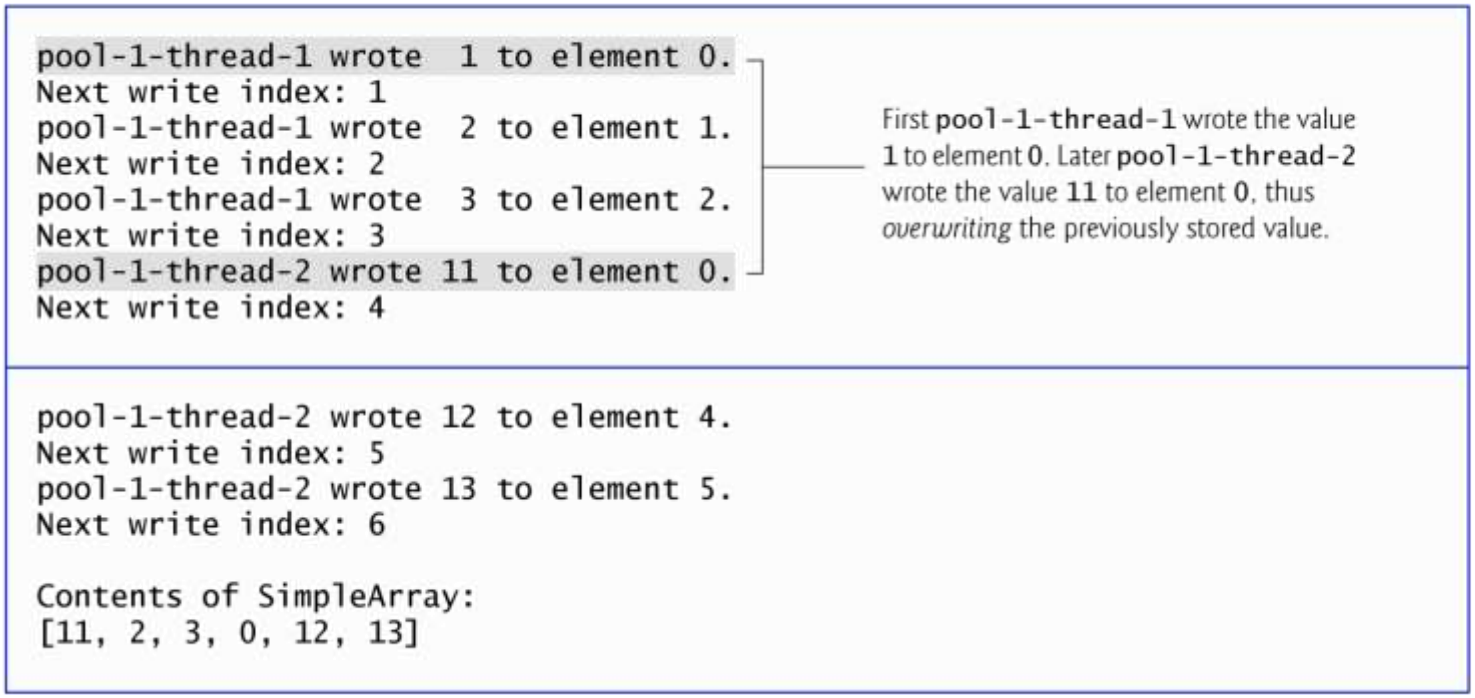


Fig. 23.7 | Executing two `Runnable`s to add elements to a shared array. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.) (Part 3 of 3.)

23.4.4 Synchronized Mutable Data Sharing—Making Operations Atomic

- The output errors of Fig. 23.7 can be attributed to the fact that the shared object, `SimpleArray`, is not thread safe.
- If one thread obtains the value of `writeIndex`, there is no guarantee that another thread cannot come along and increment `writeIndex` before the first thread has had a chance to place a value in the array.
- If this happens, the first thread will be writing to the array based on a stale value of `writeIndex`—a value that is no longer valid.

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

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

Software Engineering Observation 23.5

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

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

- Figure 23.8 displays class `SimpleArray` with the proper synchronization.
- Identical to the `SimpleArray` class of Fig. 23.5, except that `add` is now a `synchronized` method (line 20)—only one thread at a time can execute this method.
- We reuse classes `ArrayWriter` (Fig. 23.6) and `SharedArrayTest` (Fig. 23.7) from the previous example.
- We output messages from `synchronized` blocks for demonstration purposes
  - I/O *should not* be performed in `synchronized` blocks, because it's important to minimize the amount of time that an object is "locked."
- Line 27 in this example calls `Thread` method `sleep` to emphasize the unpredictability of thread scheduling.
  - *Never* call `sleep` while holding a lock in a real application.

```
1  // Fig. 23.8: SimpleArray.java
2  // Class that manages an integer array to be shared by multiple
3  // threads with synchronization.
4  import java.security.SecureRandom;
5  import java.util.Arrays;
6
7  public class SimpleArray
8  {
9     private static final SecureRandom generator = new SecureRandom();
10    private final int[] array; // the shared integer array
11    private int writeIndex = 0; // index of next element to be written
12
13    // construct a SimpleArray of a given size
14    public SimpleArray(int size)
15    {
16       array = new int[size];
17    }
18
19    // add a value to the shared array
20    public synchronized void add(int value)
21    {
22       int position = writeIndex; // store the write index
23
```

Fig. 23.8 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 1 of 3.)

```
24          try
25          {
26             // in real applications, you shouldn't sleep while holding a lock
27             Thread.sleep(generator.nextInt(500)); // for demo only
28          }
29          catch (InterruptedException ex)
30          {
31             Thread.currentThread().interrupt();
32          }
33
34          // put value in the appropriate element
35          array[position] = value;
36          System.out.printf("%s wrote %2d to element %d.%n",
37             Thread.currentThread().getName(), value, position);
38
39          ++writeIndex; // increment index of element to be written next
40          System.out.printf("Next write index: %d%n", writeIndex);
41       }
42
```

Fig. 23.8 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 2 of 3.)

```
43      // used for outputting the contents of the shared integer array
44      public synchronized String toString()
45      {
46         return Arrays.toString(array);
47      }
48   } // end class SimpleArray
```

```
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 3 of 3.)

Performance Tip 23.2

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

23.10 Concurrent Collections

- The collections from the `java.util.concurrent` package are specifically designed and optimized for sharing collections among multiple threads.
- Fig. 23.22 lists the many concurrent collections in package `java.util.concurrent`.
- For more Java SE 7 information, visit
  `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`
- For Java SE 8 information, visit
  `http://download.java.net/jdk8/docs/api/java/util/concurrent/package-summary.html`

Collection	Description
`ArrayBlockingQueue`	A fixed-size queue that supports the producer/consumer relationship—possibly with many producers and consumers.
`ConcurrentHashMap`	**A hash-based map (similar to the `HashMap` introduced in** Chapter 16**) that allows an arbitrary number of reader threads and a limited number of writer threads. This and the `LinkedBlockingQueue` are by far the most frequently used concurrent collections.**
`ConcurrentLinkedDeque`	A concurrent linked-list implementation of a double-ended queue.
`ConcurrentLinkedQueue`	A concurrent linked-list implementation of a queue that can grow dynamically.
`ConcurrentSkipListMap`	A concurrent map that is sorted by its keys.
`ConcurrentSkipListSet`	A sorted concurrent set.
`CopyOnWriteArrayList`	A thread-safe `ArrayList`. Each operation that modifies the collection first creates a new copy of the contents. Used when the collection is traversed much more frequently than the collection's contents are modified.
`CopyOnWriteArraySet`	A set that's implemented using `CopyOnWriteArrayList`.
`DelayQueue`	A variable-size queue containing `Delayed` objects. An object can be removed only after its delay has expired.

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

Collection	Description
`LinkedBlockingDeque`	A double-ended blocking queue implemented as a linked list that can optionally be fixed in size.
`LinkedBlockingQueue`	A blocking queue implemented as a linked list that can optionally be fixed in size. This and the ConcurrentHashMap are by far the most frequently used concurrent collections.
`LinkedTransferQueue`	A linked-list implementation of interface `TransferQueue`. Each producer has the option of waiting for a consumer to take an element being inserted (via method `transfer`) or simply placing the element into the queue (via method put). Also provides overloaded method `tryTransfer` to immediately transfer an element to a waiting consumer or to do so within a specified timeout period. If the transfer cannot be completed, the element is not placed in the queue. Typically used in applications that pass messages between threads.
`PriorityBlockingQueue`	A variable-length priority-based blocking queue (like a `PriorityQueue`).
`SynchronousQueue`	[For experts.] A blocking queue implementation that does not have an internal capacity. Each insert operation by one thread must wait for a remove operation from another thread and vice versa.

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