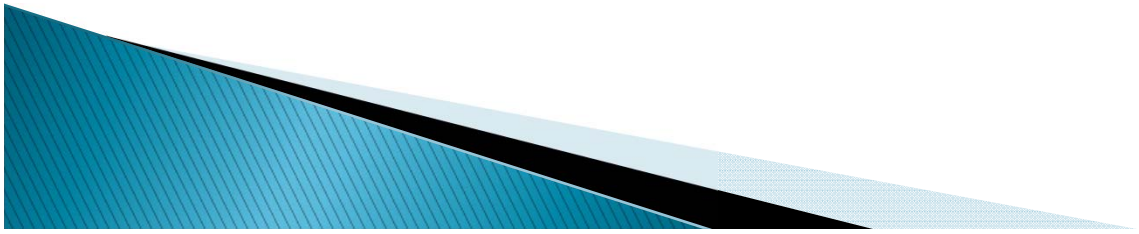
Chapter 26 Multithreading

Java How to Program, 9/e

OBJECTIVES

In this chapter you'll learn:

- What threads are and why they're useful.
- How threads enable you to manage concurrent activities.
- The life cycle of a thread.
- To create and execute `Runnable`s.
- Thread synchronization.
- What producer/consumer relationships are and how they're implemented with multithreading.
- To enable multiple threads to update Swing GUI components in a thread-safe manner.



26.1 Introduction

- Operating systems on single-processor computers create the illusion of concurrent execution by rapidly switching between activities, but on such computers only a single instruction can execute at once.
- 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 applicationwide resources such as memory with them.
- This capability is called multithreading.

Performance Tip 26.1

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 processors (if available) so that multiple tasks execute truly concurrently and the application can operate more efficiently. Multithreading can also increase performance on single-processor systems that simulate concurrency—when one thread cannot proceed (because, for example, it's waiting for the result of an I/O operation), another can use the processor.

26.1 Introduction (cont.)

- Programming concurrent applications is difficult and error prone.
- Guidelines:
  - Use existing classes from the Java API that manage synchronization for you.
  - If you need even more complex capabilities, use interfaces `Lock` and `Condition`.
  - Interfaces `Lock` and `Condition` should be used only by advanced programmers who are familiar with concurrent programming's common traps and pitfalls.

26.2 Thread States: Life Cycle of a Thread

- At any time, a thread is said to be in one of several thread states—illustrated in the UML state diagram in Fig. 26.1.
- 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.**
- 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.

26.2 Thread States: Life Cycle of a Thread (cont.)

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

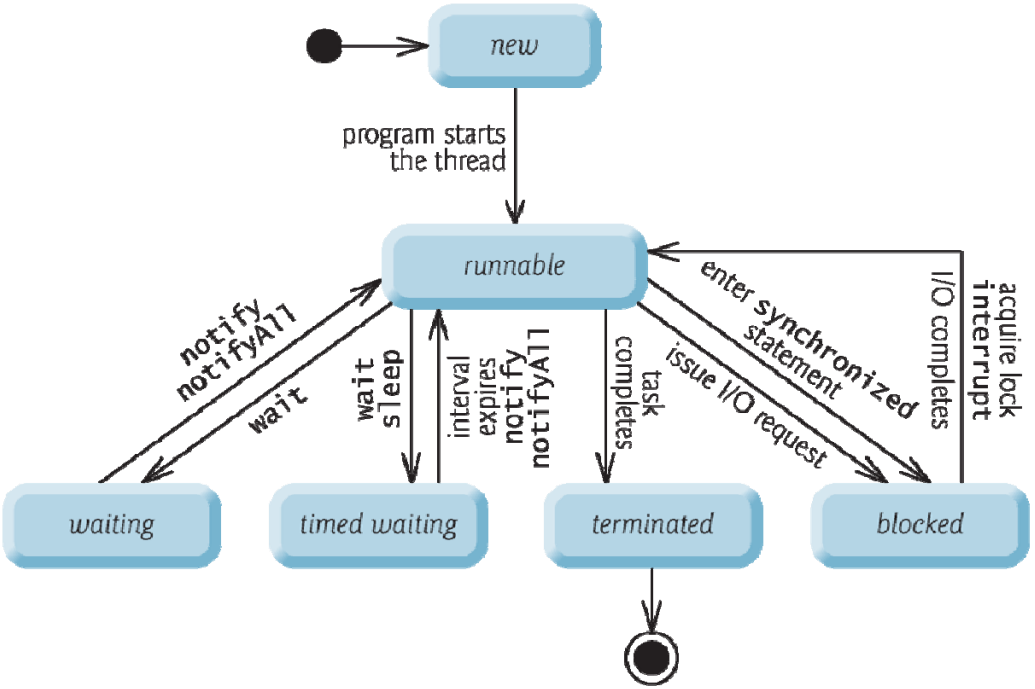


Fig. 26.1 | Thread life-cycle UML state diagram.

26.2 Thread States: Life Cycle of a Thread (cont.)

- At the operating-system level, Java's *runnable* state typically encompasses two separate states (Fig. 26.2).
- 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.

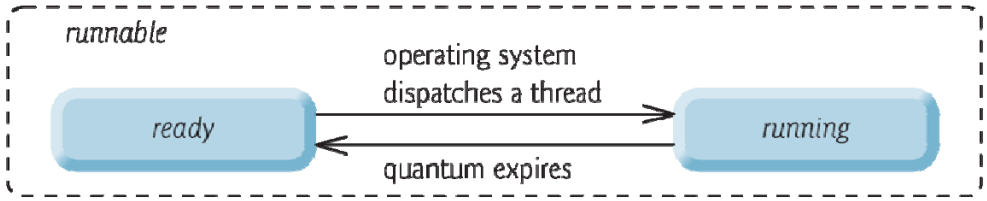


Fig. 26.2 | Operating system's internal view of Java's *runnable* state.

26.2 Thread States: Lifecycle of a Thread (cont.)

- 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 26.3).

26.2 Thread States: Lifecycle of a Thread (cont.)

- Most operating systems support timeslicing, which enables threads of equal priority to share a processor.
- An operating system's thread scheduler determines which thread runs next.
- One simple thread-scheduler implementation keeps the highest-priority thread running at all times and, if there's more than one highest-priority thread, ensures that all such threads execute for a quantum each in round-robin fashion. This process continues until all threads run to completion.

26.2 Thread States: Lifecycle of a Thread (cont.)

- When a higher-priority thread enters the *ready* state, the operating system generally preempts the currently *running* thread (an operation known as preemptive scheduling).
- Higher-priority threads could postpone—possibly indefinitely—the execution of lower-priority threads.
- Indefinite postponement is sometimes referred to as starvation.
- Operating systems employ a technique called *aging* to prevent starvation.

26.2 Thread States: Lifecycle of a Thread (cont.)

- 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 26.1

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

26.3 Creating and Executing Threads with `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 26.1

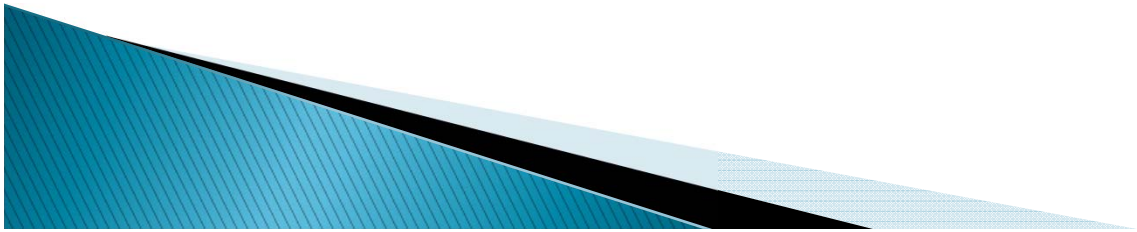
Though it's possible to create threads explicitly, it's recommended that you use the `Executor` *interface to manage the execution of* `Runnable` *objects.*

26.3 Creating and Executing Threads with `Executor` Framework (cont.)

- Class `PrintTask` (Fig. 26.3) 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.

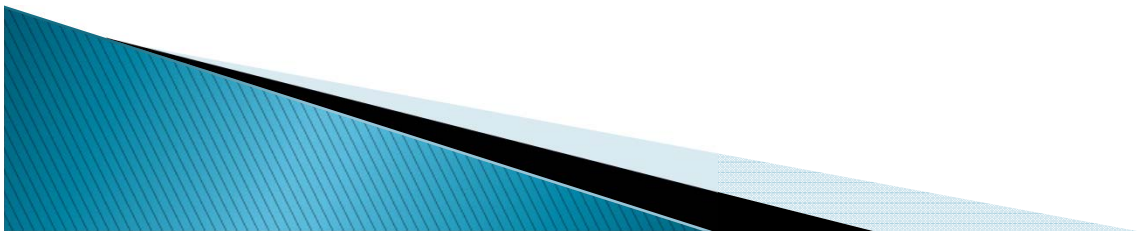
```
1  // Fig. 26.3: PrintTask.java
2  // PrintTask class sleeps for a random time from 0 to 5 seconds
3  import java.util.Random;
4
5  public class PrintTask implements Runnable
6  {
7     private final int sleepTime; // random sleep time for thread
8     private final String taskName; // name of task
9     private final static Random generator = new Random();
10
11    // constructor
12    public PrintTask( String name )
13    {
14       taskName = name; // set task name
15
16       // pick random sleep time between 0 and 5 seconds
17       sleepTime = generator.nextInt( 5000 ); // milliseconds
18    } // end PrintTask constructor
19
```

Fig. 26.3 | 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         } // end try
29         catch ( InterruptedException exception )
30         {
31            System.out.printf( "%s %s\n", taskName,
32               "terminated prematurely due to interruption" );
33         } // end catch
34
35         // print task name
36         System.out.printf( "%s done sleeping\n", taskName );
37      } // end method run
38   } // end class PrintTask
```

Fig. 26.3 | `PrintTask` class sleeps for a random time from 0 to 5 seconds. (Part 2 of 2.)



```
1  // Fig. 26.4: 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 threadExecutor = Executors.newCachedThreadPool();
19
20       // start threads and place in runnable state
21       threadExecutor.execute( task1 ); // start task1
22       threadExecutor.execute( task2 ); // start task2
23       threadExecutor.execute( task3 ); // start task3
24
```

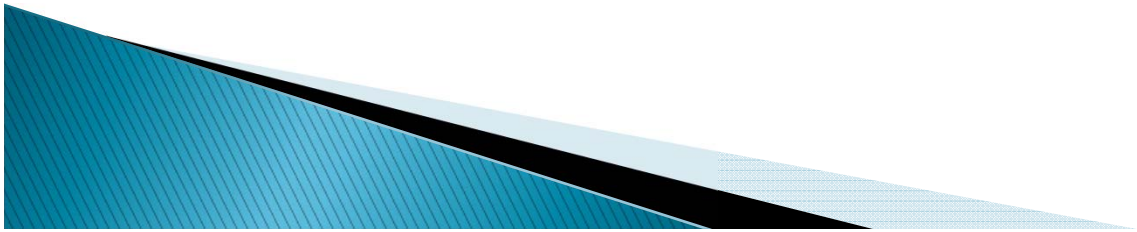
Fig. 26.4 | Using an `ExecutorService` to execute `Runnable`s. (Part 1 of 3.)

```
25          // shut down worker threads when their tasks complete
26          threadExecutor.shutdown();
27
28          System.out.println( "Tasks started, main ends.\n" );
29       } // end main
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
```

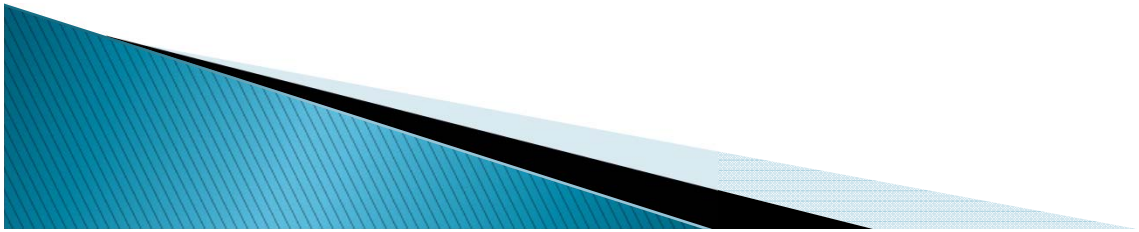
Fig. 26.4 | Using an `ExecutorService` to execute `Runnable`s. (Part 2 of 3.)



```
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. 26.4 | Using an `ExecutorService` to execute `Runnable`s. (Part 3 of 3.)



26.3 Creating and Executing Threads with `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.

26.3 Creating and Executing Threads with `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`.

26.4 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 manipulates the shared object.
  - During that time, other threads desiring to manipulate the object are kept waiting.
  - When the thread with exclusive access to the object finishes manipulating it, one of the threads that was waiting 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.

26.4 Thread Synchronization (cont.)

- 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

26.4 Thread Synchronization (cont.)

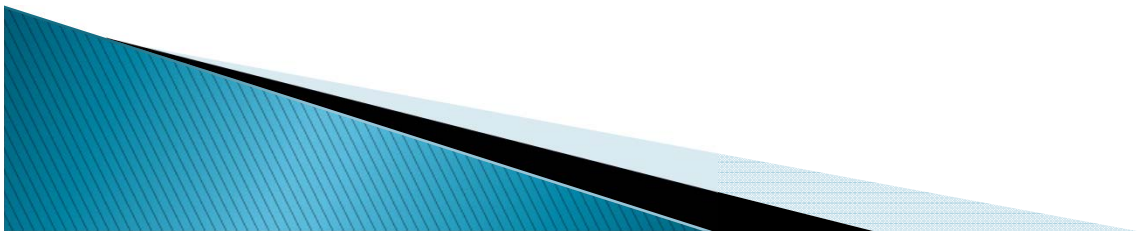
- The `synchronized` statements are declared using the `synchronized` keyword:
  - ```
    synchronized ( object )
    {
       statements
    } // end synchronized statement
    ```
- where *object* is the object whose monitor lock will be acquired
  - *object* is normally `this` if it's the object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

26.4.1 Unsynchronized Data Sharing

- A `SimpleArray` object (Fig. 26.5) 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 data more obvious.

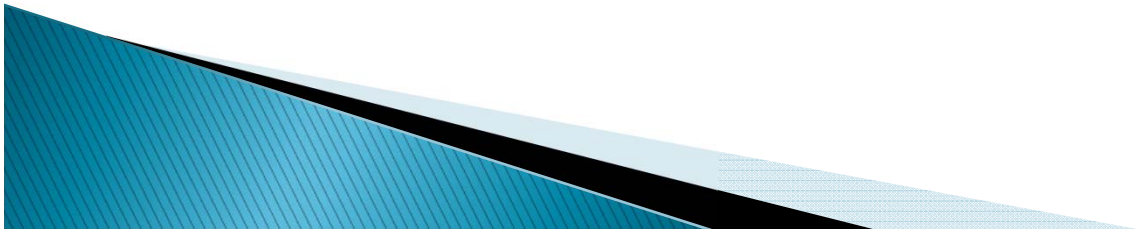
```
1  // Fig. 26.5: SimpleArray.java
2  // Class that manages an integer array to be shared by multiple threads.
3  import java.util.Arrays;
4  import java.util.Random;
5
6  public class SimpleArray // CAUTION: NOT THREAD SAFE!
7  {
8     private final int[] array; // the shared integer array
9     private int writeIndex = 0; // index of next element to be written
10    private final static Random generator = new Random();
11
12    // construct a SimpleArray of a given size
13    public SimpleArray( int size )
14    {
15       array = new int[ size ];
16    } // end constructor
17
```

Fig. 26.5 | Class that manages an integer array to be shared by multiple threads. (Part 1 of 3.)



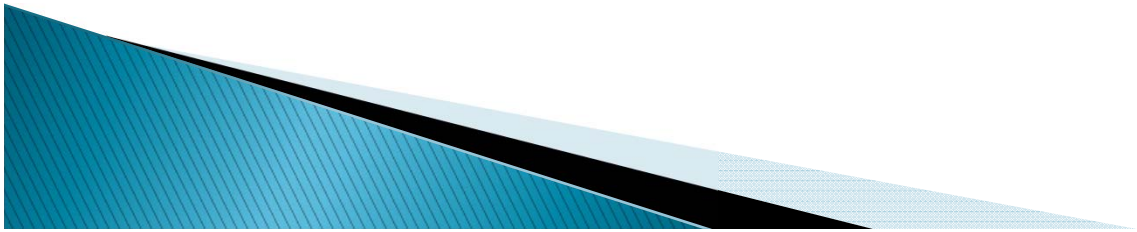
```
18      // add a value to the shared array
19      public void add( int value )
20      {
21         int position = writeIndex; // store the write index
22
23         try
24         {
25            // put thread to sleep for 0-499 milliseconds
26            Thread.sleep( generator.nextInt( 500 ) );
27         } // end try
28         catch ( InterruptedException ex )
29         {
30            ex.printStackTrace();
31         } // end catch
32
```

Fig. 26.5 | Class that manages an integer array to be shared by multiple threads. (Part 2 of 3.)



```
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       } // end method add
41
42       // used for outputting the contents of the shared integer array
43       public String toString()
44       {
45          return "\nContents of SimpleArray:\n" + Arrays.toString( array );
46       } // end method toString
47    } // end class SimpleArray
```

Fig. 26.5 | Class that manages an integer array to be shared by multiple threads. (Part 3 of 3.)

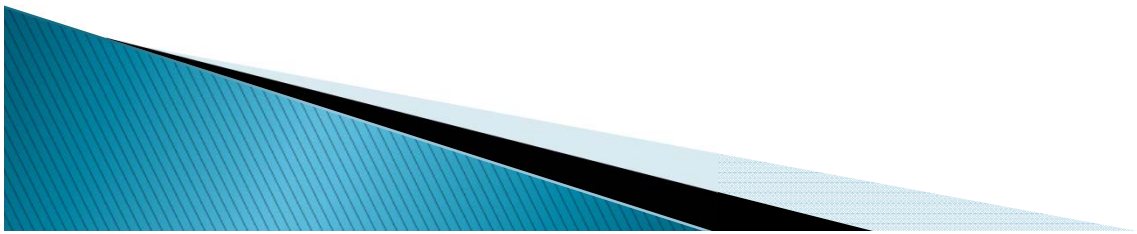


26.4.1 Unsynchronized Data Sharing (cont.)

- Class `ArrayWriter` (Fig. 26.6) 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 added to the `SimpleArray` object.

```
1  // Fig. 26.6: 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    } // end constructor
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       } // end for
22    } // end method run
23 } // end class ArrayWriter
```

Fig. 26.6 | Adds integers to an array shared with other `Runnable`s.



26.4.1 Unsynchronized Data Sharing (cont.)

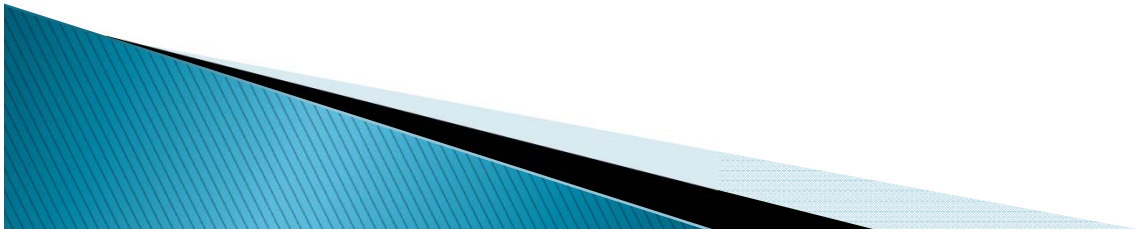
- Class `SharedArrayTest` (Fig. 26.7) executes two `ArrayWriter` tasks that add values to a single `SimpleArray` object.
- `ExecutorService`'s `shutDown` method prevents additional tasks from starting and to enable the application to terminate when the currently executing tasks complete execution.
- 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 26.7: SharedArrayTest.java
2  // Executes 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 executor = Executors.newCachedThreadPool();
20       executor.execute( writer1 );
21       executor.execute( writer2 );
22
23       executor.shutdown();
24
```

Fig. 26.7 | Executes two `Runnable`s to insert values in a shared array. (Part 1 of 3.)

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

Fig. 26.7 | Executes two `Runnable`s to insert values in a shared array. (Part 2 of 3.)



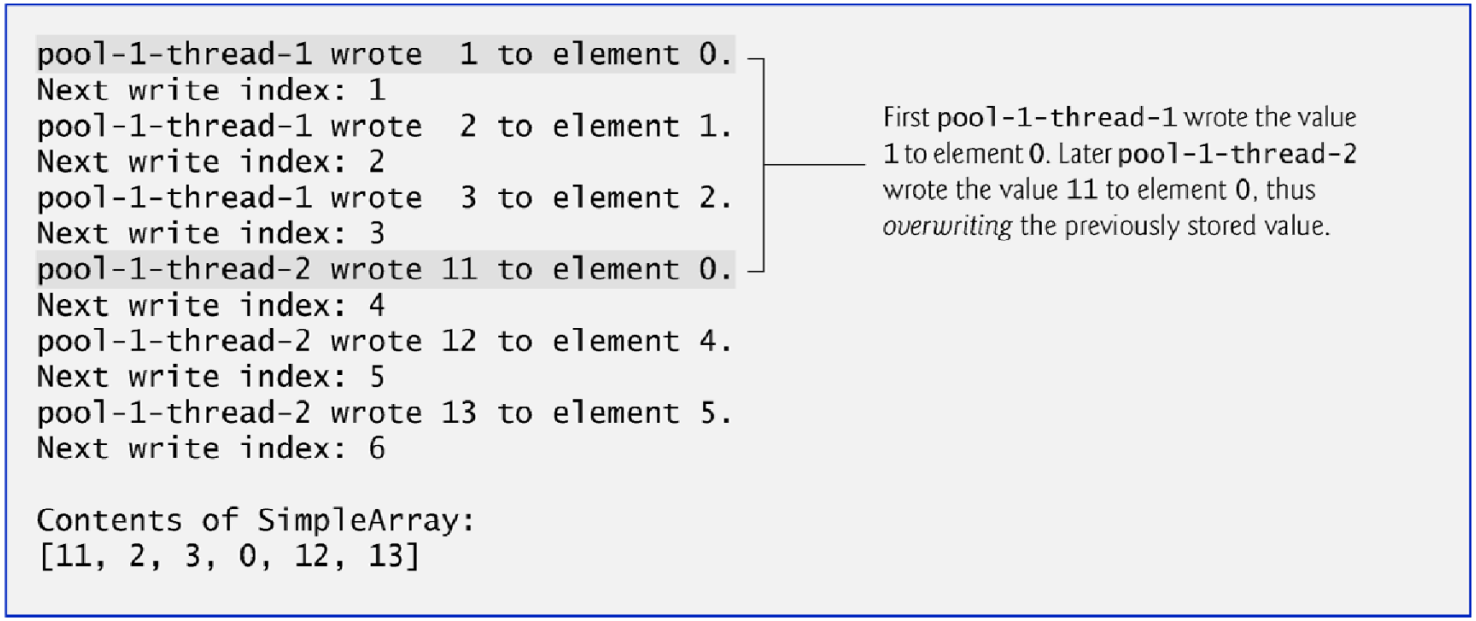


Fig. 26.7 | Executes two `Runnable`s to insert values in a shared array. (Part 3 of 3.)

26.4.2 Synchronized Data Sharing—Making Operations Atomic

- The output errors of Fig. 26.7 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.

26.4.2 Synchronized 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 26.2

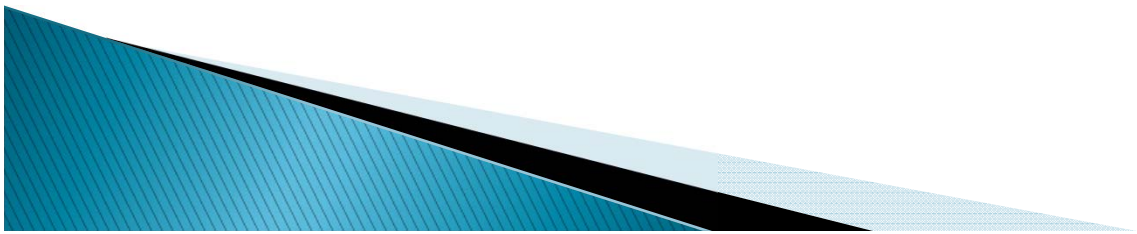
Place all accesses to mutable data that may be shared by multiple threads inside `synchronized` *statements or* `synchronized` *methods that synchronize on the same lock. When performing multiple operations on shared data, hold the lock for the entirety of the operation to ensure that the operation is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

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

- Figure 26.8 displays class `SimpleArray` with the proper synchronization.
- Identical to the `SimpleArray` class of Fig. 26.5, 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. 26.6) and `SharedArrayTest` (Fig. 26.7) 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.

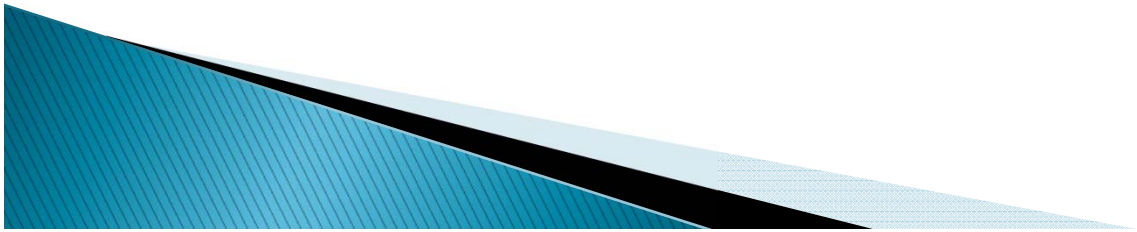
```
1  // Fig. 26.8: SimpleArray.java
2  // Class that manages an integer array to be shared by multiple
3  // threads with synchronization.
4  import java.util.Arrays;
5  import java.util.Random;
6
7  public class SimpleArray
8  {
9     private final int[] array; // the shared integer array
10    private int writeIndex = 0; // index of next element to be written
11    private final static Random generator = new Random();
12
13    // construct a SimpleArray of a given size
14    public SimpleArray( int size )
15    {
16       array = new int[ size ];
17    } // end constructor
18
```

Fig. 26.8 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 1 of 4.)



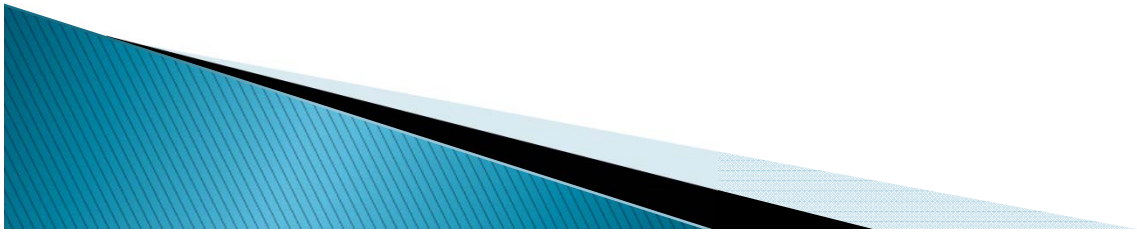
```
19      // add a value to the shared array
20      public synchronized void add( int value )
21      {
22         int position = writeIndex; // store the write index
23
24         try
25         {
26            // put thread to sleep for 0-499 milliseconds
27            Thread.sleep( generator.nextInt( 500 ) );
28         } // end try
29         catch ( InterruptedException ex )
30         {
31            ex.printStackTrace();
32         } // end catch
33
```

Fig. 26.8 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 2 of 4.)



```
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       } // end method add
42
43       // used for outputting the contents of the shared integer array
44       public String toString()
45       {
46          return "\nContents of SimpleArray:\n" + Arrays.toString( array );
47       } // end method toString
48    } // end class SimpleArray
```

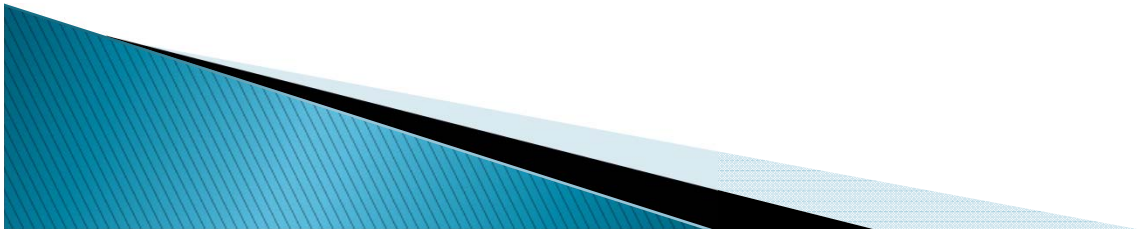
Fig. 26.8 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 3 of 4.)



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



Performance Tip 26.2

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.*

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

- Synchronization is necessary only for mutable data, or data that may change in its lifetime.
- If the shared data will not change, then it's not possible for a thread to see old or incorrect values as a result of another thread's manipulating 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.
  - Prevents accidental modification
- Labeling object references as `final` indicates that the reference will not change, but it does not guarantee that the object itself is immutable—this depends entirely on the object's properties.

Good Programming Practice 26.1

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.*