Multithreading

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.

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.

Thread States: Life Cycle of a Thread

- At any time, a thread is said to be in one of several thread 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.**
- 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.

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).

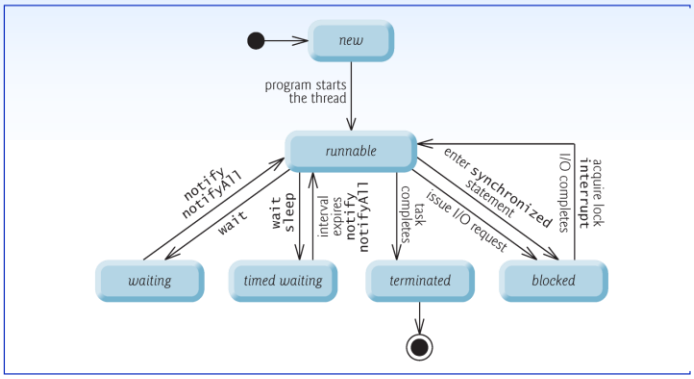

Fig. 26.1 | Thread life-cycle UML state diagram.

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

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

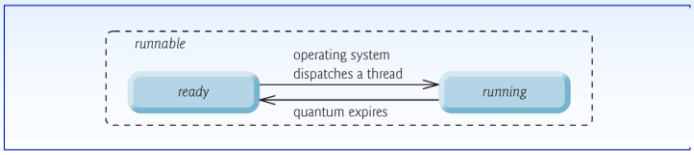

Fig. 26.2 | Operating system's internal view of Java's *runnable* state.

Thread Priorities and Thread Scheduling

- Every Java thread has a thread priority that helps determine the order in which threads are scheduled.
- Java priorities range between `MIN_PRIORITY` (a constant of 1) and `MAX_PRIORITY` (a constant of 10).
- By default, every thread is given priority `NORM_PRIORITY` (a constant of 5).
- Each new thread inherits the priority of the thread that created it.

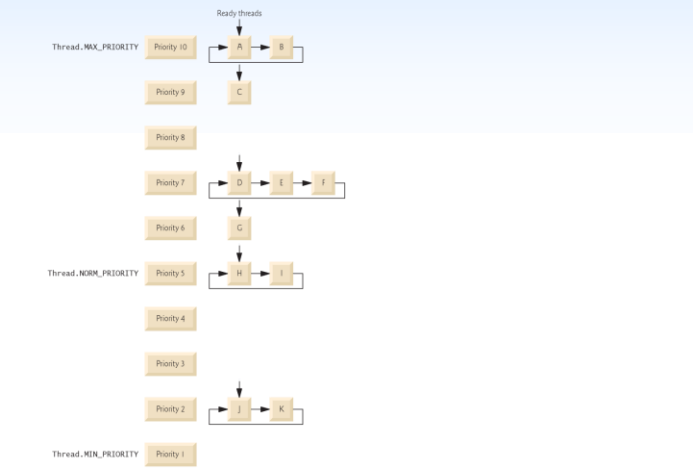

Fig. 26.3 | Thread-priority scheduling.

Portability Tip 26.1

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

Portability Tip 26.2

When designing multithreaded programs, consider the threading capabilities of all the platforms on which the programs will execute. Using priorities other than the default will make your programs' behavior platform specific. If portability is your goal, don't adjust thread priorities.

Creating and Executing Threads

- 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.

Runnables and the Thread Class

- Class `PrintTask` 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. 26.4: 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    public PrintTask( String name )
12    {
13       taskName = name; // set task name
14
15       // pick random sleep time between 0 and 5 seconds
16       sleepTime = generator.nextInt( 5000 ); // milliseconds
17    } // end PrintTask constructor
18
```

Fig. 26.4 | `PrintTask` class sleeps for a random time from 0 to 5 seconds. (Part 1 of 2.)

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

Methods called from here execute as part of the thread that calls `run`

Fig. 26.4 | `PrintTask` class sleeps for a random time from 0 to 5 seconds. (Part 2 of 2.)


```
1  // Fig. 26.5: ThreadCreator.java
2  // Creating and starting three threads to execute Runnables.
3  import java.lang.Thread;
4
5  public class ThreadCreator
6  {
7     public static void main( String[] args )
8     {
9        System.out.println( "Creating threads" );
10
11       // create each thread with a new targeted runnable
12       Thread thread1 = new Thread( new PrintTask( "task1" ) );
13       Thread thread2 = new Thread( new PrintTask( "task2" ) );
14       Thread thread3 = new Thread( new PrintTask( "task3" ) );
15
16       System.out.println( "Threads created, starting tasks." );
17
18       // start threads and place in runnable state
19       thread1.start(); // invokes task1's run method
20       thread2.start(); // invokes task2's run method
21       thread3.start(); // invokes task3's run method
22
23       System.out.println( "Tasks started, main ends.\n" );
24    } // end main
25 } // end class ThreadCreator
```

Main thread ends after this but program continues until other threads die

Fig. 26.5 | Creating and starting three threads to execute `Runnable`s. (Part 1 of 2.)

```
Creating threads
Threads created, starting tasks
Tasks started, main ends

task3 going to sleep for 491 milliseconds
task2 going to sleep for 71 milliseconds
task1 going to sleep for 3549 milliseconds
task2 done sleeping
task3 done sleeping
task1 done sleeping
```

```
Creating threads
Threads created, starting tasks
task1 going to sleep for 4666 milliseconds
task2 going to sleep for 48 milliseconds
task3 going to sleep for 3924 milliseconds
Tasks started, main ends

task2 done sleeping
task3 done sleeping
task1 done sleeping
```

Fig. 26.5 | Creating and starting three threads to execute `Runnable`s. (Part 2 of 2.)

Thread Management with the `Executor` Framework

- 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.

Thread Management 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.


```
1  // Fig. 26.6: 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
```

Fig. 26.6 | Using an `ExecutorService` to execute `Runnables`. (Part 1 of 3.)

```
24
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
```

No more threads can be started with this thread pool at this point

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


```
Starting Executor
task1 going to sleep for 1342 milliseconds
task2 going to sleep for 277 milliseconds
task3 going to sleep for 2737 milliseconds
Tasks started, main ends

task2 done sleeping
task1 done sleeping
task3 done sleeping
```

Fig. 26.6 | Using an `ExecutorService` to execute `Runnable`s. (Part 3 of 3.)

Thread Synchronization

- When multiple threads share an object and it is modified by one or more of them, indeterminate results may occurunless 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.

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

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.

Unsynchronized Data Sharing

- A `SimpleArray` object 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. 26.7: 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
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. 26.7 | Class that manages an integer array to be shared by multiple threads. (Part 1 of 3.)


```
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
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
```

Fig. 26.7 | Class that manages an integer array to be shared by multiple threads. (Part 2 of 3.)

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

Unsynchronized Data Sharing (cont.)

- Class `ArrayWriter` 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.

```java
// Fig. 26.8: ArrayWriter.java
// Adds integers to an array shared with other Runnables
import java.lang.Runnable;

public class ArrayWriter implements Runnable
{
   private final SimpleArray sharedSimpleArray;
   private final int startValue;

   public ArrayWriter( int value, SimpleArray array )
   {
      startValue = value;
      sharedSimpleArray = array;
   } // end constructor

   public void run()
   {
      for ( int i = startValue; i < startValue + 3; i++ )
      {
         sharedSimpleArray.add( i ); // add an element to the shared array
      } // end for
   } // end method run
} // end class ArrayWriter
```

Fig. 26.8 | Adds integers to an array shared with other `Runnable`s.

Unsynchronized Data Sharing (cont.)

- Class `SharedArrayTest` 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 26.9: 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();
```

Fig. 26.9 | Executes two `Runnable`s to insert values in a shared array. (Part 1 of 3.)


```
24
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. 26.9 | Executes two `Runnable`s to insert values in a shared array. (Part 2 of 3.)

```
pool-1-thread-1 wrote  1 to element 0.
Next write index: 1
pool-1-thread-1 wrote  2 to element 1.
Next write index: 2
pool-1-thread-1 wrote  3 to element 2.
Next write index: 3
pool-1-thread-2 wrote 11 to element 0.
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]
```

First `pool-1-thread-1` wrote the value 1 to element 0. Later `pool-1-thread-2` wrote the value 11 to element 0, thus overwriting the previously stored value.

Fig. 26.9 | Executes two `Runnable`s to insert values in a shared array. (Part 3 of 3.)

Synchronized Data Sharing—Making Operations Atomic

- The output errors of 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.

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 26.1

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.*

Synchronized Data Sharing—Making Operations Atomic (cont.)

- Figure displays class `SimpleArray` with the proper synchronization.
- Identical to the `SimpleArray` class, except that `add` is now a `synchronized` method (line 20)—only one thread at a time can execute this method.
- We reuse classes `ArrayWriter` and `SharedArrayTest` 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. 26.10: SimpleArray.java
2  // Class that manages an integer array to be shared by multiple threads
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. 26.10 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 1 of 3.)

```
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
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
```

Only one thread at a time can execute this method

Fig. 26.10 | Class that manages an integer array to be shared by multiple threads with synchronization. (Part 2 of 3.)


```
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
```

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

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.*

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 is 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.*

Producer/Consumer Relationship without Synchronization

- In a producer/consumer relationship, the producer portion of an application generates data and stores it in a shared object, and the consumer portion of the application reads data from the shared object.
- A producer thread generates data and places it in a shared object called a buffer.
- A consumer thread reads data from the buffer.
- This relationship requires synchronization to ensure that values are produced and consumed properly.
- Operations on the buffer data shared by a producer and consumer thread are also state dependent—the operations should proceed only if the buffer is in the correct state.
- If the buffer is in a not-full state, the producer may produce; if the buffer is in a not-empty state, the consumer may consume.

```
1  // Fig. 26.11: Buffer.java
2  // Buffer interface specifies methods called by Producer and Consumer.
3  public interface Buffer
4  {
5     // place int value into Buffer
6     public void set( int value ) throws InterruptedException;
7
8     // return int value from Buffer
9     public int get() throws InterruptedException;
10 } // end interface Buffer
```

Fig. 26.11 | Buffer interface specifies methods called by `Producer` and `Consumer`.


```
1  // Fig. 26.12: Producer.java
2  // Producer with a run method that inserts the values 1 to 10 in buffer.
3  import java.util.Random;
4
5  public class Producer implements Runnable
6  {
7     private final static Random generator = new Random();
8     private final Buffer sharedLocation; // reference to shared object
9
10    // constructor
11    public Producer( Buffer shared )
12    {
13       sharedLocation = shared;
14    } // end Producer constructor
15
16    // store values from 1 to 10 in sharedLocation
17    public void run()
18    {
19       int sum = 0;
20
```

Fig. 26.12 | `Producer` with a `run` method that inserts the values 1 to 10 in buffer. (Part 1 of 2.)

```
21         for ( int count = 1; count <= 10; count++ )
22         {
23            try // sleep 0 to 3 seconds, then place value in Buffer
24            {
25               Thread.sleep( generator.nextInt( 3000 ) ); // random sleep
26               sharedLocation.set( count ); // set value in buffer
27               sum += count; // increment sum of values
28               System.out.printf( "\t%2d\n", sum );
29            } // end try
30            // if lines 25 or 26 get interrupted, print stack trace
31            catch ( InterruptedException exception )
32            {
33               exception.printStackTrace();
34            } // end catch
35         } // end for
36
37         System.out.println(
38            "Producer done producing\nTerminating Producer" );
39      } // end method run
40   } // end class Producer
```

Fig. 26.12 | `Producer` with a `run` method that inserts the values 1 to 10 in buffer. (Part 2 of 2.)


```
1  // Fig. 26.13: Consumer.java
2  // Consumer with a run method that loops, reading 10 values from buffer.
3  import java.util.Random;
4
5  public class Consumer implements Runnable
6  {
7     private final static Random generator = new Random();
8     private final Buffer sharedLocation; // reference to shared object
9
10    // constructor
11    public Consumer( Buffer shared )
12    {
13       sharedLocation = shared;
14    } // end Consumer constructor
15
16    // read sharedLocation's value 10 times and sum the values
17    public void run()
18    {
19       int sum = 0;
20
```

Fig. 26.13 | `Consumer` with a `run` method that loops, reading 10 values from buffer. (Part 1 of 2.)

```
21         for ( int count = 1; count <= 10; count++ )
22         {
23            // sleep 0 to 3 seconds, read value from buffer and add to sum
24            try
25            {
26               Thread.sleep( generator.nextInt( 3000 ) );
27               sum += sharedLocation.get();
28               System.out.printf( "\t\t\t%2d\n", sum );
29            } // end try
30            // if lines 26 or 27 get interrupted, print stack trace
31            catch ( InterruptedException exception )
32            {
33               exception.printStackTrace();
34            } // end catch
35         } // end for
36
37         System.out.printf( "\n%s %d\n%s\n",
38            "Consumer read values totaling", sum, "Terminating Consumer" );
39      } // end method run
40   } // end class Consumer
```

Fig. 26.13 | `Consumer` with a `run` method that loops, reading 10 values from buffer. (Part 2 of 2.)


```
1  // Fig. 26.14: UnsynchronizedBuffer.java
2  // UnsynchronizedBuffer maintains the shared integer that is accessed by
3  // a producer thread and a consumer thread via methods set and get.
4  public class UnsynchronizedBuffer implements Buffer
5  {
6     private int buffer = -1; // shared by producer and consumer threads
7
8     // place value into buffer
9     public void set( int value ) throws InterruptedException
10    {
11       System.out.printf( "Producer writes\t%2d", value );
12       buffer = value;
13    } // end method set
14
15    // return value from buffer
16    public int get() throws InterruptedException
17    {
18       System.out.printf( "Consumer reads\t%2d", buffer );
19       return buffer;
20    } // end method get
21 } // end class UnsynchronizedBuffer
```

Fig. 26.14 | `UnsynchronizedBuffer` maintains the shared integer that is accessed by a producer thread and a consumer thread via methods `set` and `get`.

```
1  // Fig. 26.15: SharedBufferTest.java
2  // Application with two threads manipulating an unsynchronized buffer.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class SharedBufferTest
7  {
8     public static void main( String[] args )
9     {
10       // create new thread pool with two threads
11       ExecutorService application = Executors.newCachedThreadPool();
12
13       // create UnsynchronizedBuffer to store ints
14       Buffer sharedLocation = new UnsynchronizedBuffer();
15
16       System.out.println(
17          "Action\t\tValue\tSum of Produced\tSum of Consumed" );
18       System.out.println(
19          "------\t\t-----\t---------------\t---------------\n" );
20
```

Fig. 26.15 | Application with two threads manipulating an unsynchronized buffer. (Part 1 of 6.)


```
21       // execute the Producer and Consumer, giving each of them access
22       // to sharedLocation
23       application.execute( new Producer( sharedLocation ) );
24       application.execute( new Consumer( sharedLocation ) );
25
26       application.shutdown(); // terminate application when tasks complete
27    } // end main
28 } // end class SharedBufferTest
```

Fig. 26.15 | Application with two threads manipulating an unsynchronized buffer. (Part 2 of 6.)

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

Producer writes  1        1
Producer writes  2        3                              -- 1 is lost
Producer writes  3        6                              -- 2 is lost
Consumer reads   3                        3
Producer writes  4       10
Consumer reads   4                        7
Producer writes  5       15
Producer writes  6       21                              -- 5 is lost
Producer writes  7       28                              -- 6 is lost
Consumer reads   7                       14
Consumer reads   7                       21              -- 7 read again
Producer writes  8       36
Consumer reads   8                       29
Consumer reads   8                       37              -- 8 read again
Producer writes  9       45
Producer writes 10       55                              -- 9 is lost

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

Fig. 26.15 | Application with two threads manipulating an unsynchronized buffer. (Part 3 of 6.)


```
Consumer read values totaling 77
Terminating Consumer
```

Fig. 26.15 | Application with two threads manipulating an unsynchronized buffer. (Part 4 of 6.)

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

Consumer reads  -1                      -1 ── reads -1 bad data
Producer writes  1       1
Consumer reads   1                      0
Consumer reads   1                      1 ── 1 read again
Consumer reads   1                      2 ── 1 read again
Consumer reads   1                      3 ── 1 read again
Consumer reads   1                      4 ── 1 read again
Producer writes  2       3
Consumer reads   2                      6
Producer writes  3       6
Consumer reads   3                      9
Producer writes  4      10
Consumer reads   4                     13
Producer writes  5      15
Producer writes  6      21                ── 5 is lost
Consumer reads   6                     19

Consumer read values totaling 19
Terminating Consumer
Producer writes  7      28                ── 7 never read
Producer writes  8      36                ── 8 never read
Producer writes  9      45                ── 9 never read
Producer writes 10      55                ── 10 never read
```

Fig. 26.15 | Application with two threads manipulating an unsynchronized buffer. (Part 5 of 6.)


```
Producer done producing
Terminating Producer
```

Fig. 26.15 | Application with two threads manipulating an unsynchronized buffer. (Part 6 of 6.)

Error-Prevention Tip 26.1

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

Producer/Consumer Relationship: `ArrayBlockingQueue`

- One way to synchronize producer and consumer threads is to use classes from Java's concurrency package that encapsulate the synchronization for you.
- Java includes the class `ArrayBlockingQueue` (from package `java.util.concurrent`)—a fully implemented, thread-safe buffer class that implements interface `BlockingQueue`.
- Declares methods `put` and `take`, the blocking equivalents of `Queue` methods `offer` and `poll`, respectively.
- Method `put` places an element at the end of the `BlockingQueue`, waiting if the queue is full.
- Method `take` removes an element from the head of the `BlockingQueue`, waiting if the queue is empty.

```
1  // Fig. 26.16: BlockingBuffer.java
2  // Creating a synchronized buffer using an ArrayBlockingQueue.
3  import java.util.concurrent.ArrayBlockingQueue;
4
5  public class BlockingBuffer implements Buffer
6  {
7     private final ArrayBlockingQueue<Integer> buffer; // shared buffer
8
9     public BlockingBuffer()
10    {
11       buffer = new ArrayBlockingQueue<Integer>( 1 );
12    } // end BlockingBuffer constructor
13
14    // place value into buffer
15    public void set( int value ) throws InterruptedException
16    {
17       buffer.put( value ); // place value in buffer
18       System.out.printf( "%s%2d\t%s%d\n", "Producer writes ", value,
19          "Buffer cells occupied: ", buffer.size() );
20    } // end method set
21
```

Handles synchronization automatically

Fig. 26.16 | Creating a synchronized buffer using an `ArrayBlockingQueue`. (Part 1 of 2.)


```
22    // return value from buffer
23    public int get() throws InterruptedException
24    {
25       int readValue = buffer.take(); // remove value from buffer
26       System.out.printf( "%s %2d\t%s%d\n", "Consumer reads ",
27          readValue, "Buffer cells occupied: ", buffer.size() );
28
29       return readValue;
30    } // end method get
31 } // end class BlockingBuffer
```

Fig. 26.16 | Creating a synchronized buffer using an `ArrayBlockingQueue`. (Part 2 of 2.)

```java
// Fig. 26.17: BlockingBufferTest.java
// Two threads manipulating a blocking buffer that properly
// implements the producer/consumer relationship.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class BlockingBufferTest
{
   public static void main( String[] args )
   {
      // create new thread pool with two threads
      ExecutorService application = Executors.newCachedThreadPool();

      // create BlockingBuffer to store ints
      Buffer sharedLocation = new BlockingBuffer();

      application.execute( new Producer( sharedLocation ) );
      application.execute( new Consumer( sharedLocation ) );

      application.shutdown();
   } // end main
} // end class BlockingBufferTest
```

Fig. 26.17 | Two threads manipulating a blocking buffer that properly implements the producer/consumer relationship. (Part 1 of 2.)


```
Producer writes  1      Buffer cells occupied: 1
Consumer reads   1      Buffer cells occupied: 0
Producer writes  2      Buffer cells occupied: 1
Consumer reads   2      Buffer cells occupied: 0
Producer writes  3      Buffer cells occupied: 1
Consumer reads   3      Buffer cells occupied: 0
Producer writes  4      Buffer cells occupied: 1
Consumer reads   4      Buffer cells occupied: 0
Producer writes  5      Buffer cells occupied: 1
Consumer reads   5      Buffer cells occupied: 0
Producer writes  6      Buffer cells occupied: 1
Consumer reads   6      Buffer cells occupied: 0
Producer writes  7      Buffer cells occupied: 1
Consumer reads   7      Buffer cells occupied: 0
Producer writes  8      Buffer cells occupied: 1
Consumer reads   8      Buffer cells occupied: 0
Producer writes  9      Buffer cells occupied: 1
Consumer reads   9      Buffer cells occupied: 0
Producer writes 10      Buffer cells occupied: 1

Producer done producing
Terminating Producer
Consumer reads  10      Buffer cells occupied: 0

Consumer read values totaling 55
Terminating Consumer
```

Fig. 26.17 | Two threads manipulating a blocking buffer that properly implements the producer/consumer relationship. (Part 2 of 2.)

Producer/Consumer Relationship with Synchronization

- For educational purposes, we now explain how you can implement a shared buffer yourself using the `synchronized` keyword and methods of class `Object`.
- The first step in synchronizing access to the buffer is to implement methods `get` and `set` as `synchronized` methods.
- This requires that a thread obtain the monitor lock on the `Buffer` object before attempting to access the buffer data.

Producer/Consumer Relationship with Synchronization (cont.)

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

Common Programming Error 26.1

It's an error if a thread issues a `wait`*, a* `notify` *or a* `notifyAll` *on an object without having acquired a lock for it. This causes an* `IllegalMonitorStateException`*.*

Error-Prevention Tip 26.2

It's a good practice to use `notifyAll` *to notify* waiting *threads to become* runnable*. Doing so avoids the possibility that your program would forget about waiting threads, which would otherwise starve.*

```
1  // Fig. 26.18: SynchronizedBuffer.java
2  // Synchronizing access to shared data using Object
3  // methods wait and notifyAll.
4  public class SynchronizedBuffer implements Buffer
5  {
6     private int buffer = -1; // shared by producer and consumer threads
7     private boolean occupied = false; // whether the buffer is occupied
8
9     // place value into buffer
10    public synchronized void set( int value ) throws InterruptedException
11    {
12       // while there are no empty locations, place thread in waiting state
13       while ( occupied )
14       {
15          // output thread information and buffer information, then wait
16          System.out.println( "Producer tries to write." );
17          displayState( "Buffer full. Producer waits." );
18          wait();
19       } // end while
20
21       buffer = value; // set new buffer value
22
```

Fig. 26.18 | Synchronizing access to shared data using `Object` methods `wait` and `notifyAll`. (Part 1 of 3.)


```
23       // indicate producer cannot store another value
24       // until consumer retrieves current buffer value
25       occupied = true;
26
27       displayState( "Producer writes " + buffer );
28
29       notifyAll(); // tell waiting thread(s) to enter runnable state
30    } // end method set; releases lock on SynchronizedBuffer
31
32    // return value from buffer
33    public synchronized int get() throws InterruptedException
34    {
35       // while no data to read, place thread in waiting state
36       while ( !occupied )
37       {
38          // output thread information and buffer information, then wait
39          System.out.println( "Consumer tries to read." );
40          displayState( "Buffer empty. Consumer waits." );
41          wait();
42       } // end while
43
```

Fig. 26.18 | Synchronizing access to shared data using `Object` methods `wait` and `notifyAll`. (Part 2 of 3.)

```
44          // indicate that producer can store another value
45          // because consumer just retrieved buffer value
46          occupied = false;
47
48          displayState( "Consumer reads " + buffer );
49
50          notifyAll(); // tell waiting thread(s) to enter runnable state
51
52          return buffer;
53       } // end method get; releases lock on SynchronizedBuffer
54
55       // display current operation and buffer state
56       public void displayState( String operation )
57       {
58          System.out.printf( "%-40s%d\t\t%b\n\n", operation, buffer,
59             occupied );
60       } // end method displayState
61    } // end class SynchronizedBuffer
```

Fig. 26.18 | Synchronizing access to shared data using `Object` methods `wait` and `notifyAll`. (Part 3 of 3.)

Producer/Consumer Relationship with Synchronization (cont.)

- Class `SynchronizedBuffer`'s `occupied` field is used to determine whether it's the `Producer`'s or the `Consumer`'s turn to perform a task.
- This field is used in conditional expressions in both the `set` and `get` methods.
- If `occupied` is `false`, then `buffer` is empty, so the `Consumer` cannot read the value of `buffer`, but the `Producer` can place a value into `buffer`.
- If `occupied` is `true`, the `Consumer` can read a value from `buffer`, but the `Producer` cannot place a value into `buffer`.

Error-Prevention Tip 26.3

Always invoke method `wait` *in a loop that tests the condition the task is waiting on. It's possible that a thread will reenter the* runnable *state (via a timed wait or another thread calling* `notifyAll`*) before the condition is satisfied. Testing the condition again ensures that the thread will not erroneously execute if it was notified early.*


```
1  // Fig. 26.19: SharedBufferTest2.java
2  // Two threads correctly manipulating a synchronized buffer.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class SharedBufferTest2
7  {
8     public static void main( String[] args )
9     {
10       // create a newCachedThreadPool
11       ExecutorService application = Executors.newCachedThreadPool();
12
13       // create SynchronizedBuffer to store ints
14       Buffer sharedLocation = new SynchronizedBuffer();
15
16       System.out.printf( "%-40s%s\t\t%s\n%-40s%s\n\n", "Operation",
17          "Buffer", "Occupied", "---------", "------\t\t--------" );
18
19       // execute the Producer and Consumer tasks
20       application.execute( new Producer( sharedLocation ) );
21       application.execute( new Consumer( sharedLocation ) );
22
23       application.shutdown();
24    } // end main
25 } // end class SharedBufferTest2
```

Fig. 26.19 | Two threads correctly manipulating a synchronized buffer. (Part 1 of 4.)

```
Operation                          Buffer         Occupied
---------                          ------         --------

Consumer tries to read.
Buffer empty. Consumer waits.      -1             false

Producer writes 1                  1              true

Consumer reads 1                   1              false

Consumer tries to read.
Buffer empty. Consumer waits.      1              false

Producer writes 2                  2              true

Consumer reads 2                   2              false

Producer writes 3                  3              true

Consumer reads 3                   3              false

Producer writes 4                  4              true

Producer tries to write.
Buffer full. Producer waits.       4              true
```

Fig. 26.19 | Two threads correctly manipulating a synchronized buffer. (Part 2 of 4.)


```
Consumer reads 4                   4              false

Producer writes 5                  5              true

Consumer reads 5                   5              false

Producer writes 6                  6              true

Producer tries to write.
Buffer full. Producer waits.       6              true

Consumer reads 6                   6              false

Producer writes 7                  7              true

Producer tries to write.
Buffer full. Producer waits.       7              true

Consumer reads 7                   7              false

Producer writes 8                  8              true

Consumer reads 8                   8              false

Consumer tries to read.
Buffer empty. Consumer waits.      8              false
```

Fig. 26.19 | Two threads correctly manipulating a synchronized buffer. (Part 3 of 4.)

```
Producer writes 9                       9           true

Consumer reads 9                        9           false

Consumer tries to read.
Buffer empty. Consumer waits.           9           false

Producer writes 10                      10          true

Consumer reads 10                       10          false


Producer done producing
Terminating Producer

Consumer read values totaling 55
Terminating Consumer
```

Fig. 26.19 | Two threads correctly manipulating a synchronized buffer. (Part 4 of 4.)

Producer/Consumer Relationship: Bounded Buffers

- The program may not perform optimally.
- If the two threads operate at different speeds, one them will spend more (or most) of its time waiting.
- Even when we have threads that operate at the same relative speeds, those threads may occasionally become "out of sync" over a period of time, causing one of them to wait for the other.
- *We cannot make assumptions about the relative speeds of concurrent threads.*
- When threads wait excessively, programs become less efficient, interactive programs become less responsive and applications suffer longer delays.
- To minimize the amount of waiting time for threads that share resources and operate at the same average speeds, we can implement a bounded buffer that provides a fixed number of buffer cells into which the `Producer` can place values, and from which the `Consumer` can retrieve those values.

Performance Tip 26.3

Even when using a bounded buffer, it's possible that a producer thread could fill the buffer, which would force the producer to wait until a consumer consumed a value to free an element in the buffer. Similarly, if the buffer is empty at any given time, a consumer thread must wait until the producer produces another value. The key to using a bounded buffer is to optimize the buffer size to minimize the amount of thread wait time, while not wasting space.

Producer/Consumer Relationship: Bounded Buffers (cont.)

- The simplest way to implement a bounded buffer is to use an `ArrayBlockingQueue` for the buffer so that *all of the synchronization details are handled for you*.
  - This can be done by reusing previous example and simply passing the desired size for the bounded buffer into the `ArrayBlockingQueue` constructor.

Producer/Consumer Relationship: Bounded Buffers (cont.)

- Implementing Your Own Bounded Buffer as a Circular Buffer
  - The program in and demonstrates a `Producer` and a `Consumer` accessing a bounded buffer with synchronization.
- We implement the bounded buffer in class `CircularBuffer` as a circular buffer that writes into and reads from the array elements in order, beginning at the first cell and moving toward the last.
- When a `Producer` or `Consumer` reaches the last element, it returns to the first and begins writing or reading, respectively, from there.


```
1  // Fig. 26.20: CircularBuffer.java
2  // Synchronizing access to a shared three-element bounded buffer.
3  public class CircularBuffer implements Buffer
4  {
5     private final int[] buffer = { -1, -1, -1 }; // shared buffer
6
7     private int occupiedCells = 0; // count number of buffers used
8     private int writeIndex = 0; // index of next element to write to
9     private int readIndex = 0; // index of next element to read
10
11    // place value into buffer
12    public synchronized void set( int value ) throws InterruptedException
13    {
14       // wait until buffer has space avaialble, then write value;
15       // while no empty locations, place thread in blocked state
16       while ( occupiedCells == buffer.length )
17       {
18          System.out.printf( "Buffer is full. Producer waits.\n" );
19          wait(); // wait until a buffer cell is free
20       } // end while
21
22       buffer[ writeIndex ] = value; // set new buffer value
```

Fig. 26.20 | Synchronizing access to a shared three-element bounded buffer. (Part 1 of 4.)

```
23
24          // update circular write index
25          writeIndex = ( writeIndex + 1 ) % buffer.length;
26
27          ++occupiedCells; // one more buffer cell is full
28          displayState( "Producer writes " + value );
29          notifyAll(); // notify threads waiting to read from buffer
30       } // end method set
31
32       // return value from buffer
33       public synchronized int get() throws InterruptedException
34       {
35          // wait until buffer has data, then read value;
36          // while no data to read, place thread in waiting state
37          while ( occupiedCells == 0 )
38          {
39             System.out.printf( "Buffer is empty. Consumer waits.\n" );
40             wait(); // wait until a buffer cell is filled
41          } // end while
42
43          int readValue = buffer[ readIndex ]; // read value from buffer
44
```

Fig. 26.20 | Synchronizing access to a shared three-element bounded buffer. (Part 2 of 4.)


```
45          // update circular read index
46          readIndex = ( readIndex + 1 ) % buffer.length;
47
48          --occupiedCells; // one fewer buffer cells are occupied
49          displayState( "Consumer reads " + readValue );
50          notifyAll(); // notify threads waiting to write to buffer
51
52          return readValue;
53       } // end method get
54
55       // display current operation and buffer state
56       public void displayState( String operation )
57       {
58          // output operation and number of occupied buffer cells
59          System.out.printf( "%s%s%d)\n%s", operation,
60             " (buffer cells occupied: ", occupiedCells, "buffer cells:  " );
61
62          for ( int value : buffer )
63             System.out.printf( " %2d  ", value ); // output values in buffer
64
65          System.out.print( "\n               " );
66
```

Fig. 26.20 | Synchronizing access to a shared three-element bounded buffer. (Part 3 of 4.)

```
67         for ( int i = 0; i < buffer.length; i++ )
68            System.out.print( "---- " );
69
70         System.out.print( "\n               " );
71
72         for ( int i = 0; i < buffer.length; i++ )
73         {
74            if ( i == writeIndex && i == readIndex )
75               System.out.print( " WR" ); // both write and read index
76            else if ( i == writeIndex )
77               System.out.print( " W   " ); // just write index
78            else if ( i == readIndex )
79               System.out.print( "  R  " ); // just read index
80            else
81               System.out.print( "     " ); // neither index
82         } // end for
83
84         System.out.println( "\n" );
85      } // end method displayState
86   } // end class CircularBuffer
```

Fig. 26.20 | Synchronizing access to a shared three-element bounded buffer. (Part 4 of 4.)


```
1  // Fig. 26.21: CircularBufferTest.java
2  // Producer and Consumer threads manipulating a circular buffer.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class CircularBufferTest
7  {
8     public static void main( String[] args )
9     {
10       // create new thread pool with two threads
11       ExecutorService application = Executors.newCachedThreadPool();
12
13       // create CircularBuffer to store ints
14       CircularBuffer sharedLocation = new CircularBuffer();
15
16       // display the initial state of the CircularBuffer
17       sharedLocation.displayState( "Initial State" );
18
19       // execute the Producer and Consumer tasks
20       application.execute( new Producer( sharedLocation ) );
21       application.execute( new Consumer( sharedLocation ) );
22
```

Fig. 26.21 | `Producer` and `Consumer` threads manipulating a circular buffer. (Part 1 of 6.)

```
        application.shutdown();
    } // end main
} // end class CircularBufferTest
```

```
Initial State (buffer cells occupied: 0)
buffer cells:   -1   -1   -1
               ---- ---- ----
                WR

Producer writes 1 (buffer cells occupied: 1)
buffer cells:    1   -1   -1
               ---- ---- ----
                 R   W

Consumer reads 1 (buffer cells occupied: 0)
buffer cells:    1   -1   -1
               ---- ---- ----
                     WR

Buffer is empty. Consumer waits.
Producer writes 2 (buffer cells occupied: 1)
buffer cells:    1    2   -1
               ---- ---- ----
                      R   W
```

Fig. 26.21 | `Producer` and `Consumer` threads manipulating a circular buffer. (Part 2 of 6.)


```
Consumer reads 2 (buffer cells occupied: 0)
buffer cells:    1    2   -1
               ---- ---- ----
                         WR

Producer writes 3 (buffer cells occupied: 1)
buffer cells:    1    2    3
               ---- ---- ----
                W          R

Consumer reads 3 (buffer cells occupied: 0)
buffer cells:    1    2    3
               ---- ---- ----
                WR

Producer writes 4 (buffer cells occupied: 1)
buffer cells:    4    2    3
               ---- ---- ----
                 R   W

Producer writes 5 (buffer cells occupied: 2)
buffer cells:    4    5    3
               ---- ---- ----
                 R        W
```

Fig. 26.21 | `Producer` and `Consumer` threads manipulating a circular buffer. (Part 3 of 6.)

```
Consumer reads 4 (buffer cells occupied: 1)
buffer cells:    4    5    3
                ---- ---- ----
                      R   W

Producer writes 6 (buffer cells occupied: 2)
buffer cells:    4    5    6
                ---- ---- ----
                 W     R

Producer writes 7 (buffer cells occupied: 3)
buffer cells:    7    5    6
                ---- ---- ----
                     WR

Consumer reads 5 (buffer cells occupied: 2)
buffer cells:    7    5    6
                 ---- ---- ----
                     W     R

Producer writes 8 (buffer cells occupied: 3)
buffer cells:    7    8    6
                ---- ---- ----
                          WR
```

Fig. 26.21 | `Producer` and `Consumer` threads manipulating a circular buffer. (Part 4 of 6.)


```
Consumer reads 6 (buffer cells occupied: 2)
buffer cells:    7    8    6
                ---- ---- ----
                  R        W

Consumer reads 7 (buffer cells occupied: 1)
buffer cells:    7    8    6
                ---- ---- ----
                       R   W

Producer writes 9 (buffer cells occupied: 2)
buffer cells:    7    8    9
                ---- ---- ----
                 W     R

Consumer reads 8 (buffer cells occupied: 1)
buffer cells:    7    8    9
                ---- ---- ----
                 W          R

Consumer reads 9 (buffer cells occupied: 0)
buffer cells:    7    8    9
                ---- ---- ----
                 WR
```

Fig. 26.21 | `Producer` and `Consumer` threads manipulating a circular buffer. (Part 5 of 6.)

```
Producer writes 10 (buffer cells occupied: 1)
buffer cells:   10    8    9
                ---- ---- ----
                  R   W


Producer done producing
Terminating Producer
Consumer reads 10 (buffer cells occupied: 0)
buffer cells:   10    8    9
                ---- ---- ----
                     WR


Consumer read values totaling: 55
Terminating Consumer
```

Fig. 26.21 | `Producer` and `Consumer` threads manipulating a circular buffer. (Part 6 of 6.)

Producer/Consumer Relationship: The `Lock` and `Condition` Interfaces

- In this section, we discuss the `Lock` and `Condition` interfaces, which were introduced in Java SE 5.
- These interfaces give you more precise control over thread synchronization, but are more complicated to use.
- Any object can contain a reference to an object that implements the `Lock` interface (of package `java.util.concurrent.locks`).
- A thread calls the `Lock`'s `lock` method to acquire the lock.
- Once a `Lock` has been obtained by one thread, the `Lock` object will not allow another thread to obtain the `Lock` until the first thread releases the `Lock` (by calling the `Lock`'s `unlock` method).
- If several threads are trying to call method `lock` on the same `Lock` object at the same time, only one of these threads can obtain the lock—all the others are placed in the *waiting state for that lock.*

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

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

Software Engineering Observation 26.2

Using a `ReentrantLock` with a fairness policy avoids indefinite postponement.

Performance Tip 26.4

Using a `ReentrantLock` *with a fairness policy can decrease program performance significantly.*

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

- If a thread that owns a `Lock` determines that it cannot continue with its task until some condition is satisfied, the thread can wait on a condition object.
  - Allows you to explicitly declare the condition objects on which a thread may need to wait.
- For example, in the producer/consumer relationship, producers can wait on one object and consumers can wait on another.
  - Not possible when using the `synchronized` keywords and an object's built-in monitor lock.
- Condition objects are associated with a specific `Lock` and are created by calling a `Lock`'s `newCondition` method, which returns an object that implements the `Condition` interface (of package `java.util.concurrent.locks`).

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

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

Common Programming Error 26.2

Deadlock occurs when a waiting thread (let's call this thread1) cannot proceed because it's waiting (either directly or indirectly) for another thread (let's call this thread2) to proceed, while simultaneously thread2 cannot proceed because it's waiting (either directly or indirectly) for thread1 to proceed. The two threads are waiting for each other, so the actions that would enable each thread to continue execution can never occur.

Error-Prevention Tip 26.4

When multiple threads manipulate a shared object using locks, ensure that if one thread calls method `await` *to enter the* waiting *state for a condition object, a separate thread eventually will call* `Condition` *method* `signal` *to transition the thread waiting on the condition object back to the* runnable *state. If multiple threads may be waiting on the condition object, a separate thread can call* `Condition` *method* `signalAll` *as a safeguard to ensure that all the waiting threads have another opportunity to perform their tasks. If this is not done, starvation might occur.*

Common Programming Error 26.3

An `IllegalMonitorStateException` *occurs if a thread issues an* `await`*, a* `signal`*, or a* `signalAll` *on a* `Condition` *object that was created from a* `ReentrantLock` *without having acquired the lock for that* `Condition` *object.*

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

- `Lock`s allow you to interrupt waiting threads or to specify a time-out for waiting to acquire a lock, which is not possible using the `synchronized` keyword.
- Also, a `Lock` is not constrained to be acquired and released in the same block of code.
- `Condition` objects allow you to specify multiple conditions on which threads may wait.
- With the `synchronized` keyword, there is no way to explicitly state the condition on which threads are waiting.

Error-Prevention Tip 26.5

Using interfaces `Lock` *and* `Condition` *is error prone—*`unlock` *is not guaranteed to be called, whereas the monitor in a* `synchronized` *statement will always be released when the statement completes execution.*

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

- Lines 14–15 create two `Conditions` using `Lock` method `newCondition`.
- `Condition canWrite` contains a queue for a `Producer` thread waiting while the buffer is full.
  - If the buffer is full, the `Producer` calls method `await` on this `Condition`.
  - When the `Consumer` reads data from a full buffer, it calls method `signal` on this `Condition`.
- `Condition canRead` contains a queue for a `Consumer` thread waiting while the buffer is empty (i.e., there is no data in the buffer for the `Consumer` to read).
  - If the buffer is empty, the `Consumer` calls method `await` on this `Condition`.
  - When the `Producer` writes to the empty buffer, it calls method `signal` on this `Condition`.


```
1  // Fig. 26.22: SynchronizedBuffer.java
2  // Synchronizing access to a shared integer using the Lock and Condition
3  // interfaces
4  import java.util.concurrent.locks.Lock;
5  import java.util.concurrent.locks.ReentrantLock;
6  import java.util.concurrent.locks.Condition;
7
8  public class SynchronizedBuffer implements Buffer
9  {
10    // Lock to control synchronization with this buffer
11    private final Lock accessLock = new ReentrantLock();
12
13    // conditions to control reading and writing
14    private final Condition canWrite = accessLock.newCondition();
15    private final Condition canRead = accessLock.newCondition();
16
17    private int buffer = -1; // shared by producer and consumer threads
18    private boolean occupied = false; // whether buffer is occupied
19
20    // place int value into buffer
21    public void set( int value ) throws InterruptedException
22    {
```

Fig. 26.22 | Synchronizing access to a shared integer using the `Lock` and `Condition` interfaces. (Part 1 of 5.)

```
23          accessLock.lock(); // lock this object
24
25          // output thread information and buffer information, then wait
26          try
27          {
28             // while buffer is not empty, place thread in waiting state
29             while ( occupied )
30             {
31                System.out.println( "Producer tries to write." );
32                displayState( "Buffer full. Producer waits." );
33                canWrite.await(); // wait until buffer is empty
34             } // end while
35
36             buffer = value; // set new buffer value
37
38             // indicate producer cannot store another value
39             // until consumer retrieves current buffer value
40             occupied = true;
41
42             displayState( "Producer writes " + buffer );
43
```

Fig. 26.22 | Synchronizing access to a shared integer using the `Lock` and `Condition` interfaces. (Part 2 of 5.)


```
44             // signal thread waiting to read from buffer
45             canRead.signal();
46          } // end try
47          finally
48          {
49             accessLock.unlock(); // unlock this object
50          } // end finally
51       } // end method set
52
53       // return value from buffer
54       public int get() throws InterruptedException
55       {
56          int readValue = 0; // initialize value read from buffer
57          accessLock.lock(); // lock this object
58
59          // output thread information and buffer information, then wait
60          try
61          {
62             // if there is no data to read, place thread in waiting state
63             while ( !occupied )
64             {
65                System.out.println( "Consumer tries to read." );
```

Fig. 26.22 | Synchronizing access to a shared integer using the `Lock` and `Condition` interfaces. (Part 3 of 5.)

```
66              displayState( "Buffer empty. Consumer waits." );
67              canRead.await(); // wait until buffer is full
68           } // end while
69
70           // indicate that producer can store another value
71           // because consumer just retrieved buffer value
72           occupied = false;
73
74           readValue = buffer; // retrieve value from buffer
75           displayState( "Consumer reads " + readValue );
76
77           // signal thread waiting for buffer to be empty
78           canWrite.signal();
79        } // end try
80        finally
81        {
82           accessLock.unlock(); // unlock this object
83        } // end finally
84
85        return readValue;
86     } // end method get
87
```

Fig. 26.22 | Synchronizing access to a shared integer using the `Lock` and `Condition` interfaces. (Part 4 of 5.)


```
88     // display current operation and buffer state
89     public void displayState( String operation )
90     {
91        System.out.printf( "%-40s%d\t\t%b\n\n", operation, buffer,
92           occupied );
93     } // end method displayState
94  } // end class SynchronizedBuffer
```

Fig. 26.22 | Synchronizing access to a shared integer using the `Lock` and `Condition` interfaces. (Part 5 of 5.)

Error-Prevention Tip 26.6

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

Common Programming Error 26.4

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

```java
// Fig. 26.23: SharedBufferTest2.java
// Two threads manipulating a synchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SharedBufferTest2
{
   public static void main( String[] args )
   {
      // create new thread pool with two threads
      ExecutorService application = Executors.newCachedThreadPool();

      // create SynchronizedBuffer to store ints
      Buffer sharedLocation = new SynchronizedBuffer();

      System.out.printf( "%-40s%s\t\t%s\n%-40s%s\n\n", "Operation",
         "Buffer", "Occupied", "---------", "------\t\t--------" );

      // execute the Producer and Consumer tasks
      application.execute( new Producer( sharedLocation ) );
      application.execute( new Consumer( sharedLocation ) );

      application.shutdown();
   } // end main
} // end class SharedBufferTest2
```

Fig. 26.23 | Two threads manipulating a synchronized buffer. (Part 1 of 4.)


```
Operation                               Buffer          Occupied
---------                               ------          --------

Producer writes 1                       1               true

Producer tries to write.
Buffer full. Producer waits.            1               true

Consumer reads 1                        1               false

Producer writes 2                       2               true

Producer tries to write.
Buffer full. Producer waits.            2               true

Consumer reads 2                        2               false

Producer writes 3                       3               true

Consumer reads 3                        3               false

Producer writes 4                       4               true

Consumer reads 4                        4               false
```

Fig. 26.23 | Two threads manipulating a synchronized buffer. (Part 2 of 4.)

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

Producer writes 5                      5               true

Consumer reads 5                       5               false

Consumer tries to read.
Buffer empty. Consumer waits.          5               false

Producer writes 6                      6               true

Consumer reads 6                       6               false

Producer writes 7                      7               true

Consumer reads 7                       7               false

Producer writes 8                      8               true

Consumer reads 8                       8               false

Producer writes 9                      9               true

Consumer reads 9                       9               false
```

Fig. 26.23 | Two threads manipulating a synchronized buffer. (Part 3 of 4.)


```
Producer writes 10                     10              true


Producer done producing
Terminating Producer
Consumer reads 10                      10              false


Consumer read values totaling 55
Terminating Consumer
```

Fig. 26.23 | Two threads manipulating a synchronized buffer. (Part 4 of 4.)