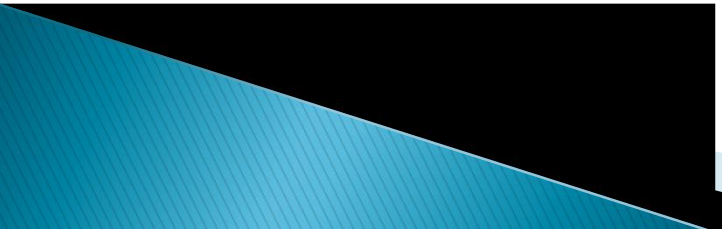
Chapter 23
Concurrency

Objectives

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

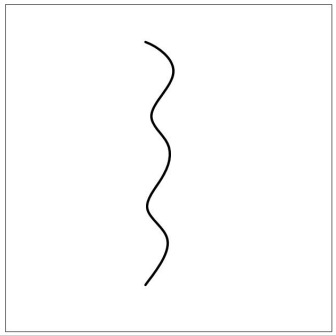
23.1 Introduction

- When we say that two tasks are operating **in parallel**, we mean that they're executing *simultaneously*.
- In this sense, parallelism is a subset of concurrency.
- Java makes concurrency available to you through the language and APIs.

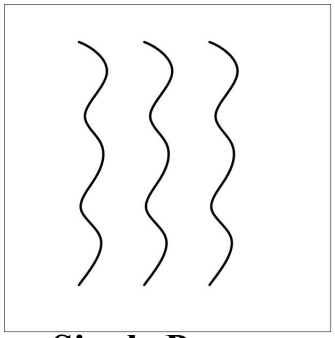


23.1 Introduction

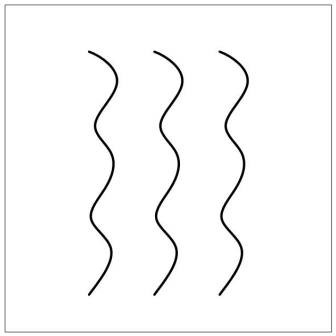
- Multithreading
  - Provides application with multiple threads of execution
  - Allows programs to perform tasks concurrently
  - Often requires programmer to synchronize threads to function correctly

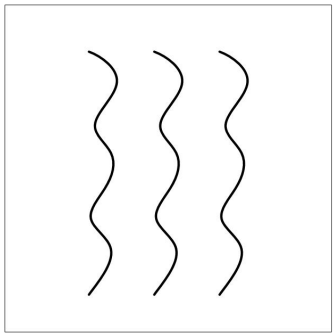


Single Process
Single Thread



Single Process
Multiple Threads





Multiple Processes
Multiple Threads

Threads and Processes

23.2 Thread States: Life Cycle of a Thread

- new **state**
  - A new thread begins its life cycle in this state**.**
  - Remains there until started, which places it in the runnable **state—considered to be executing its task.**
- *waiting* state
  - A *runnable* thread can transition to the this state while it waits for another thread to perform a task.
  - Transitions back to the runnable state only when another thread notifies it to continue executing.

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

- *timed waiting* state
  - A *runnable* thread can enter this 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.
- *blocked* state
  - A *runnable* thread transitions to this state when it attempts to perform a task that cannot be completed immediately and it must temporarily wait until that task completes.
- *terminated* state
  - A *runnable* thread enters this state when it successfully completes its task or otherwise terminates (perhaps due to an error).

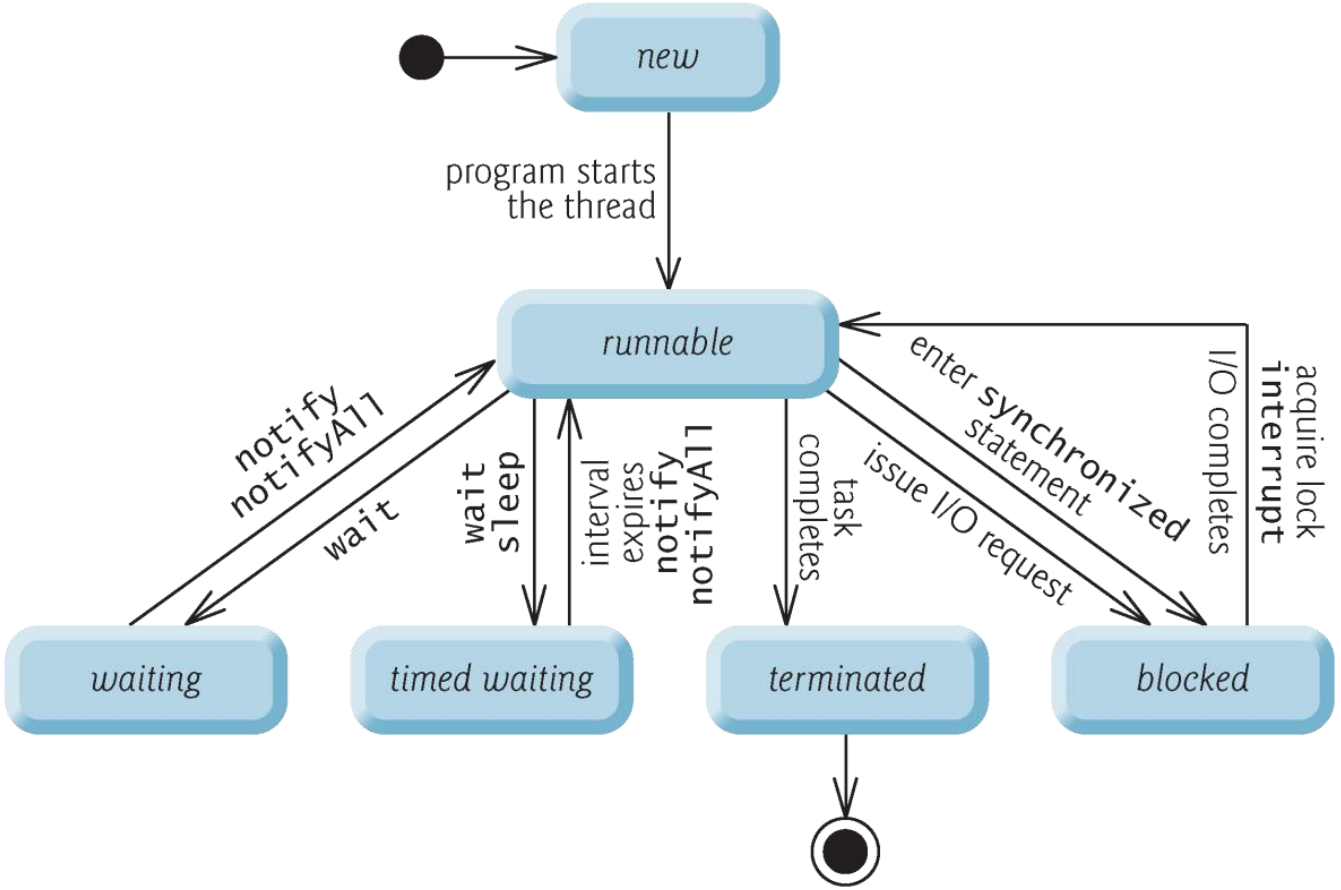


Fig. 26.1 UML state diagram.

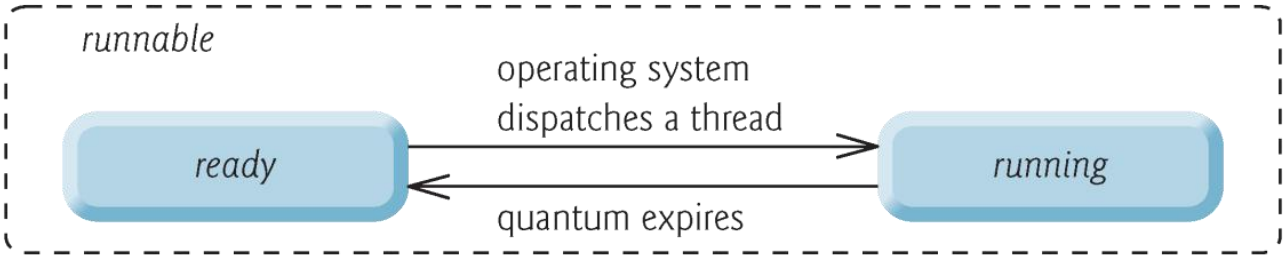


Fig. 26.2 | Operating system's internal view of Java's *runnable* state.

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

- Every Java thread has a thread priority that helps determine the order in which threads are scheduled.
- Thread priorities cannot guarantee the order in which threads execute.
  - Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10)
  - Threads with a higher priority are more important and will be allocated a processor before threads with a lower priority
  - Default priority is NORM_PRIORITY (a constant of 5)

Portability Tip 26.1

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

23.3 Creating and Executing Threads with `Executor` Framework

- Runnable interface
  - Preferred means of creating a multithreaded application
  - Declares method run
  - Executed by an object that implements the Executor interface
- Executor interface
  - Declares method execute
  - Creates and manages a group of threads called a thread pool

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

- ExecutorService interface
  - Subinterface of Executor that declares other methods for managing the life cycle of an Executor
  - Can be created using static methods of class Executors
  - Method shutdown ends threads when tasks are completed
- Executors class
  - Method newFixedThreadPool creates a pool consisting of a fixed number of threads
  - Method newCachedThreadPool creates a pool that creates new threads as they are needed

固定个数的线程池

不固定个数的线程池

Software Engineering Observation 26.1

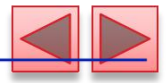
Though it's possible to create threads explicitly, it's recommended that you use the `Executor` *interface to manage the execution of* `Runnable` *objects.*

```
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 | ...ervice to execute Runnables. (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.)

23.4 Thread Synchronization

线程同步

- Thread synchronization

不是所有共享数据都需要

  - Provided to the programmer with mutual exclusion(互斥)
  - Exclusive access to a shared object， during that time, other threads desiring to access the object are kept waiting.
  - Implemented in Java using locks
- Lock interface
  - lock method obtains the lock, enforcing mutual exclusion
  - unlock method releases the lock
  - Class implements the Lock interface

Immutable Data

不变或者可读的内容

- Actually, thread synchronization is necessary only for shared mutable 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

简单的

wait、notify是object的方法
notifyAll：告诉等待进程可以开始了

23.4 Thread Synchronization (cont.)

- The `synchronized` statements are declared using the synchronized keyword:
  - `synchronized ( object )`
    `{`
    `statements`
    `} // end synchronized statement`

另一个线程要锁着

对数据区域加锁

线程访问数据时，考虑锁的问题

- 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.

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

23.4.1 Unsynchronized Data Sharing

SimpleArray is *not thread safe because it allows any number of threads to read and modify shared mutable data concurrently, which can cause errors*

```
1  // Fig. 23.5: SimpleArray.java
2  // Class that manages an integer array to be shared by multiple threads.
3  import java.security.SecureRandom;
4  import java.util.Arrays;
5
6  public class SimpleArray // CAUTION: NOT THREAD SAFE!
7  {
8     private static final SecureRandom generator = new SecureRandom();
9     private final int[] array; // the shared integer array
10    private int writeIndex = 0; // shared index of next element to write
11
```

```
12      // construct a SimpleArray of a given size
13      public SimpleArray(int size)
14      {
15         array = new int[size];
16      }
17
18      // add a value to the shared array
19      public void add(int value)
20      {
21         int position = writeIndex; // store the write index
22
23         try
24         {
25            // put thread to sleep for 0-499 milliseconds
26            Thread.sleep(generator.nextInt(500));
27         }
28         catch (InterruptedException ex)
29         {
30            Thread.currentThread().interrupt(); // re-interrupt the thread
31         }
32
33         // put value in the appropriate element
34         array[position] = value;
35         System.out.printf("%s wrote %2d to element %d.%n",
36            Thread.currentThread().getName(), value, position);
37
38         ++writeIndex; // increment index of element to be written next
39         System.out.printf("Next write index: %d%n", writeIndex);
40      }
41
42      // used for outputting the contents of the shared integer array
43      public String toString()
44      {
45         return Arrays.toString(array);
46      }
47   } // end class SimpleArray
```

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

```
1  // Fig. 23.7: SharedArrayTest.java
2  // Executing two Runnables to add elements to a shared SimpleArray.
3  import java.util.concurrent.Executors;
4  import java.util.concurrent.ExecutorService;
5  import java.util.concurrent.TimeUnit;
6
7  public class SharedArrayTest
8  {
9     public static void main(String[] arg)
10    {
11       // construct the shared object
12       SimpleArray sharedSimpleArray = new SimpleArray(6);
13
14       // create two tasks to write to the shared SimpleArray
15       ArrayWriter writer1 = new ArrayWriter(1, sharedSimpleArray);
16       ArrayWriter writer2 = new ArrayWriter(11, sharedSimpleArray);
17
18       // execute the tasks with an ExecutorService
19       ExecutorService executorService = Executors.newCachedThreadPool();
20       executorService.execute(writer1);
21       executorService.execute(writer2);
22
23       executorService.shutdown();
24
```

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

```
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
```

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

Interface ExecutorService

- shutdown()
  - Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.

- awaitTermination(long timeout, TimeUnit unit)
  - Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

Class SimpleArray with Synchronization

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

```
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            // in real applications, you shouldn't sleep while holding a lock
27            Thread.sleep(generator.nextInt(500)); // for demo only
28         }
29         catch (InterruptedException ex)
30         {
31            Thread.currentThread().interrupt();
32         }
33
34         // put value in the appropriate element
35         array[position] = value;
36         System.out.printf("%s wrote %2d to element %d.%n",
37            Thread.currentThread().getName(), value, position);
38
39         ++writeIndex; // increment index of element to be written next
40         System.out.printf("Next write index: %d%n", writeIndex);
41      }
42
43      // used for outputting the contents of the shared integer array
44      public synchronized String toString()
45      {
46         return Arrays.toString(array);
47      }
48   } // end class SimpleArray
```

```
pool-1-thread-1 wrote  1 to element 0.
Next write index: 1
pool-1-thread-2 wrote 11 to element 1.
Next write index: 2
pool-1-thread-2 wrote 12 to element 2.
Next write index: 3
pool-1-thread-2 wrote 13 to element 3.
Next write index: 4
pool-1-thread-1 wrote  2 to element 4.
Next write index: 5
pool-1-thread-1 wrote  3 to element 5.
Next write index: 6

Contents of SimpleArray:
[1, 11, 12, 13, 2, 3]
```

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

- Synchronization is necessary only for mutable data, or data that may change in its lifetime.
- 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.*

23.5 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. 23.9: 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 blockingPut(int value) throws InterruptedException;
7
8     // return int value from Buffer
9     public int blockingGet() throws InterruptedException;
10 } // end interface Buffer
```

```
1  // Fig. 23.10: Producer.java
2  // Producer with a run method that inserts the values 1 to 10 in buffer.
3  import java.security.SecureRandom;
4
5  public class Producer implements Runnable
6  {
7     private static final SecureRandom generator = new SecureRandom();
8     private final Buffer sharedLocation; // reference to shared object
9
10    // constructor
11    public Producer(Buffer sharedLocation)
12    {
13       this.sharedLocation = sharedLocation;
14    }
```

```
16      // store values from 1 to 10 in sharedLocation
17      public void run()
18      {
19         int sum = 0;
20
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.blockingPut(count); // set value in buffer
27               sum += count; // increment sum of values
28               System.out.printf("\t%2d%n", sum);
29            }
30            catch (InterruptedException exception)
31            {
32               Thread.currentThread().interrupt();
33            }
34         }
35
36         System.out.printf(
37            "Producer done producing%nTerminating Producer%n");
38      }
39   } // end class Producer
```

```
1  // Fig. 23.11: Consumer.java
2  // Consumer with a run method that loops, reading 10 values from buffer.
3  import java.security.SecureRandom;
4
5  public class Consumer implements Runnable
6  {
7     private static final SecureRandom generator = new SecureRandom();
8     private final Buffer sharedLocation; // reference to shared object
9
10    // constructor
11    public Consumer(Buffer sharedLocation)
12    {
13       this.sharedLocation = sharedLocation;
14    }
```

```
16       // read sharedLocation's value 10 times and sum the values
17       public void run()
18       {
19          int sum = 0;
20
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.blockingGet();
28                System.out.printf("\t\t\t%2d%n", sum);
29             }
30             catch (InterruptedException exception)
31             {
32                Thread.currentThread().interrupt();
33             }
34          }
35
36          System.out.printf("%n%s %d%n%s%n",
37             "Consumer read values totaling", sum, "Terminating Consumer");
38       }
39    } // end class Consumer
```

```
1  // Fig. 23.12: UnsynchronizedBuffer.java
2  // UnsynchronizedBuffer maintains the shared integer that is accessed by
3  // a producer thread and a consumer thread.
4  public class UnsynchronizedBuffer implements Buffer
5  {
6     private int buffer = -1; // shared by producer and consumer threads
7
8     // place value into buffer
9     public void blockingPut(int value) throws InterruptedException
10    {
11       System.out.printf("Producer writes\t%2d", value);
12       buffer = value;
13    }
14
15    // return value from buffer
16    public int blockingGet() throws InterruptedException
17    {
18       System.out.printf("Consumer reads\t%2d", buffer);
19       return buffer;
20    }
21 } // end class UnsynchronizedBuffer
```

```java
// Fig. 23.13: SharedBufferTest.java
// Application with two threads manipulating an unsynchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

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

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

      System.out.println(
         "Action\t\tValue\tSum of Produced\tSum of Consumed");
      System.out.printf(
         "------\t\t-----\t---------------\t---------------%n%n");

      // execute the Producer and Consumer, giving each
      // access to the sharedLocation
      executorService.execute(new Producer(sharedLocation));
      executorService.execute(new Consumer(sharedLocation));

      executorService.shutdown(); // terminate app when tasks complete
      executorService.awaitTermination(1, TimeUnit.MINUTES);
   }
} // end class SharedBufferTest
```

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
Consumer read values totaling 77				
Terminating Consumer				

Error-Prevention Tip 26.1

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

23.6 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.

```java
// Fig. 23.14: BlockingBuffer.java
// Creating a synchronized buffer using an ArrayBlockingQueue.
import java.util.concurrent.ArrayBlockingQueue;

public class BlockingBuffer implements Buffer
{
   private final ArrayBlockingQueue<Integer> buffer; // shared buffer

   public BlockingBuffer()
   {
      buffer = new ArrayBlockingQueue<Integer>(1);
   }

   // place value into buffer
   public void blockingPut(int value) throws InterruptedException
   {
      buffer.put(value); // place value in buffer
      System.out.printf("%s%2d\t%s%d%n", "Producer writes ", value,
         "Buffer cells occupied: ", buffer.size());
   }

   // return value from buffer
   public int blockingGet() throws InterruptedException
   {
      int readValue = buffer.take(); // remove value from buffer
      System.out.printf("%s %2d\t%s%d%n", "Consumer reads ",
         readValue, "Buffer cells occupied: ", buffer.size());

      return readValue;
   }
} // end class BlockingBuffer
```

```
1  // Fig. 23.15: BlockingBufferTest.java
2  // Two threads manipulating a blocking buffer that properly
3  // implements the producer/consumer relationship.
4  import java.util.concurrent.ExecutorService;
5  import java.util.concurrent.Executors;
6  import java.util.concurrent.TimeUnit;
7
8  public class BlockingBufferTest
9  {
10    public static void main(String[] args) throws InterruptedException
11    {
12       // create new thread pool with two threads
13       ExecutorService executorService = Executors.newCachedThreadPool();
14
15       // create BlockingBuffer to store ints
16       Buffer sharedLocation = new BlockingBuffer();
17
18       executorService.execute(new Producer(sharedLocation));
19       executorService.execute(new Consumer(sharedLocation));
20
21       executorService.shutdown();
22       executorService.awaitTermination(1, TimeUnit.MINUTES);
23    }
24 } // end class BlockingBufferTest
```

```
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
```

23.7 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.

23.7 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. 26.16: 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
```

while循环来判断是否状态满足，不满足则wait，应对多个线程的情况

Fig. 26.16 | Synchronizing access to shared data using `Object` methods `wait` and `notifyAll`. (Part 1 of 4.)

```
21          buffer = value; // set new buffer value
22
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
```

Fig. 26.16 | Synchronizing access to shared data using `Object` methods `wait` and `notifyAll`. (Part 2 of 4.)

```
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
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
```

Fig. 26.16 | Synchronizing access to shared data using `Object` methods `wait` and `notifyAll`. (Part 3 of 4.)

```
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. 26.16 | Synchronizing access to shared data using `Object` methods `wait` and `notifyAll`. (Part 4 of 4.)

```
1  // Fig. 26.17: 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
```

Fig. 26.17 | Two threads correctly manipulating a synchronized buffer. (Part 1 of 5.)

```
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. 26.17 | Two threads correctly manipulating a synchronized buffer. (Part 2 of 5.)

```
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. 26.17 | Two threads correctly manipulating a synchronized buffer. (Part 3 of 5.)

```
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. 26.17 ating a synchronized buffer. (Part 4 of 5.)

```
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. 26.17 | Two threads correctly manipulating a synchronized buffer. (Part 5 of 5.)

23.8 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 the example from Section 23.6 and simply passing the desired size for the bounded buffer into the `ArrayBlockingQueue` constructor.

```
1  // Fig. 26.18: 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 available, 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
```

Fig. 26.18 | Synchronizing access to a shared three-element bounded buffer. (Part 1 of 5.)

```
22          buffer[ writeIndex ] = value; // set new buffer value
23
24          // update circular write index
25          writeIndex = ( writeIndex + 1 ) % buffer.length;
26
27          ++occupiedCells; // one more buffer cell is full
28          displayState( "Producer writes " + value );
29          notifyAll(); // notify threads waiting to read from buffer
30       } // 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
```

Fig. 26.18 | Synchronizing access to a shared three-element bounded buffer. (Part 2 of 5.)

```
43          int readValue = buffer[ readIndex ]; // read value from buffer
44
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
```

Fig. 26.18 | Synchronizing access to a shared three-element bounded buffer. (Part 3 of 5.)

```
62          for ( int value : buffer )
63             System.out.printf( " %2d  ", value ); // output values in buffer
64
65          System.out.print( "\n               " );
66
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
```

Fig. 26.18 | Synchronizing access to a shared three-element bounded buffer. (Part 4 of 5.)

```
84          System.out.println( "\n" );
85       } // end method displayState
86    } // end class CircularBuffer
```

Fig. 26.18 | Synchronizing access to a shared three-element bounded buffer. (Part 5 of 5.)

```
1  // Fig. 26.19: 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
```

Fig. 26.19 | `Producer` and `Consumer` threads manipulating a circular buffer. (Part 1 of 7.)

```
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 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
```

Fig. 26.19 | `Producer` and `Consumer` threads manipulating a circular buffer. (Part 2 of 7.)

```
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

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

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

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

Fig. 26.19 | Producer and Consumer threads manipulating a circular buffer. (Part 3 of 7.)

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

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

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

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

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

Fig. 26.19 | `Producer` and `Consumer` threads manipulating a circular buffer. (Part 4

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

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

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

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

Fig. 26.19 | `Producer` and `Consumer` threads manipulating a circular buffer. (Part 5 of 7.)

```
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

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

Fig. 26.19 | `Producer` and `Consumer` threads manipulating a circular buffer. (Part 6 of 7.)

```
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. 26.19 | `Producer` and `Consumer` threads manipulating a circular buffer. (Part 7 of 7.)

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

- In this section, we discuss the `Lock` and `Condition` interfaces.
- 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.*

23.9 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 26.3

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.*

23.9 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.
- 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`).

23.9 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`.
- 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.*

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

- Locks 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.*

```
1  // Fig. 26.20: SynchronizedBuffer.java
2  // Synchronizing access to a shared integer using the Lock and Condition
3  // interfaces
4  import java.util.concurrent.locks.Lock;
5  import java.util.concurrent.locks.ReentrantLock;
6  import java.util.concurrent.locks.Condition;
7
8  public class SynchronizedBuffer implements Buffer
9  {
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    {
23       accessLock.lock(); // lock this object
```

申请获得锁

Fig. 26.20 ...s to a shared integer using the `Lock` and `Condition` interfaces

```
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
44             // signal any threads waiting to read from buffer
45             canRead.signalAll();
46          } // end try
```

唤醒canRead（写完后唤醒可读）
与上面的canWrite对应

Fig. 26.20 a shared integer using the Lock and Condition

```
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." );
66                displayState( "Buffer empty. Consumer waits." );
67                canRead.await(); // wait until buffer is full
68             } // end while
69
```

Fig. 26.20 | Synchronizing access to a shared integer using the `Lock` and `Condition` interfaces. (Part 3 of 5.)

```
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 any threads waiting for buffer to be empty
78            canWrite.signalAll();
79         } // end try
80         finally
81         {
82            accessLock.unlock(); // unlock this object
83         } // end finally
84
85         return readValue;
86      } // end method get
```

Fig. 26.20 | Synchronizing access to a shared integer using the `Lock` and `Condition` interfaces. (Part 4 of 5.)

```
87
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. 26.20 | 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.*

```
1  // Fig. 26.21: SharedBufferTest2.java
2  // Two threads 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 new thread pool with two threads
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. 26.21 ynchronized 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
```

Fig. 26.21 | Two threads manipulating a synchronized buffer. (Part 2 of 4.)

```
Producer writes 4                       4               true

Consumer reads 4                        4               false

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
```

Fig. 26.21 | Two threads manipulating a synchronized buffer. (Part 3 of 4.)

```
Producer writes 9                       9               true

Consumer reads 9                        9               false

Producer writes 10                      10              true


Producer done producing
Terminating Producer
Consumer reads 10                       10              false


Consumer read values totaling 55
Terminating Consumer
```

Fig. 26.21 | Two threads manipulating a synchronized buffer. (Part 4 of 4.)

23.10 Concurrent Collections Overview

- The collections from the `java.util.concurrent` package are specifically designed and optimized for use in programs that share collections among multiple threads.

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

Fig. 26.22 | Concurrent collections summary (package java.

Collection	Description
`LinkedBlockingDeque`	A double-ended blocking queue implemented as a linked list that can optionally be fixed in size.
`LinkedBlockingQueue`	A blocking queue implemented as a linked list that can optionally be fixed in size.
`PriorityBlockingQueue`	A variable-length priority-based blocking queue (like a `PriorityQueue`).
`SynchronousQueue`	A blocking queue implementation that does not have an internal capacity. Each insert operation by one thread must wait for a remove operation from another thread and vice versa.

Fig. 26.22 | Concurrent collections summary (package `java.util.concurrent`).

Collection	Description
Concurrent Collections Added in Java SE 7	
`ConcurrentLinkedDeque`	A concurrent linked-list implementation of a double-ended queue.
`LinkedTransferQueue`	A linked-list implementation of interface `TransferQueue`. Each producer has the option of waiting for a consumer to take an element being inserted (via method `transfer`) or simply placing the element into the queue (via method put). Also provides overloaded method `tryTransfer` to immediately transfer an element to a waiting consumer or to do so within a specified timeout period. If the transfer cannot be completed, the element is not placed in the queue. Typically used in applications that pass messages between threads.

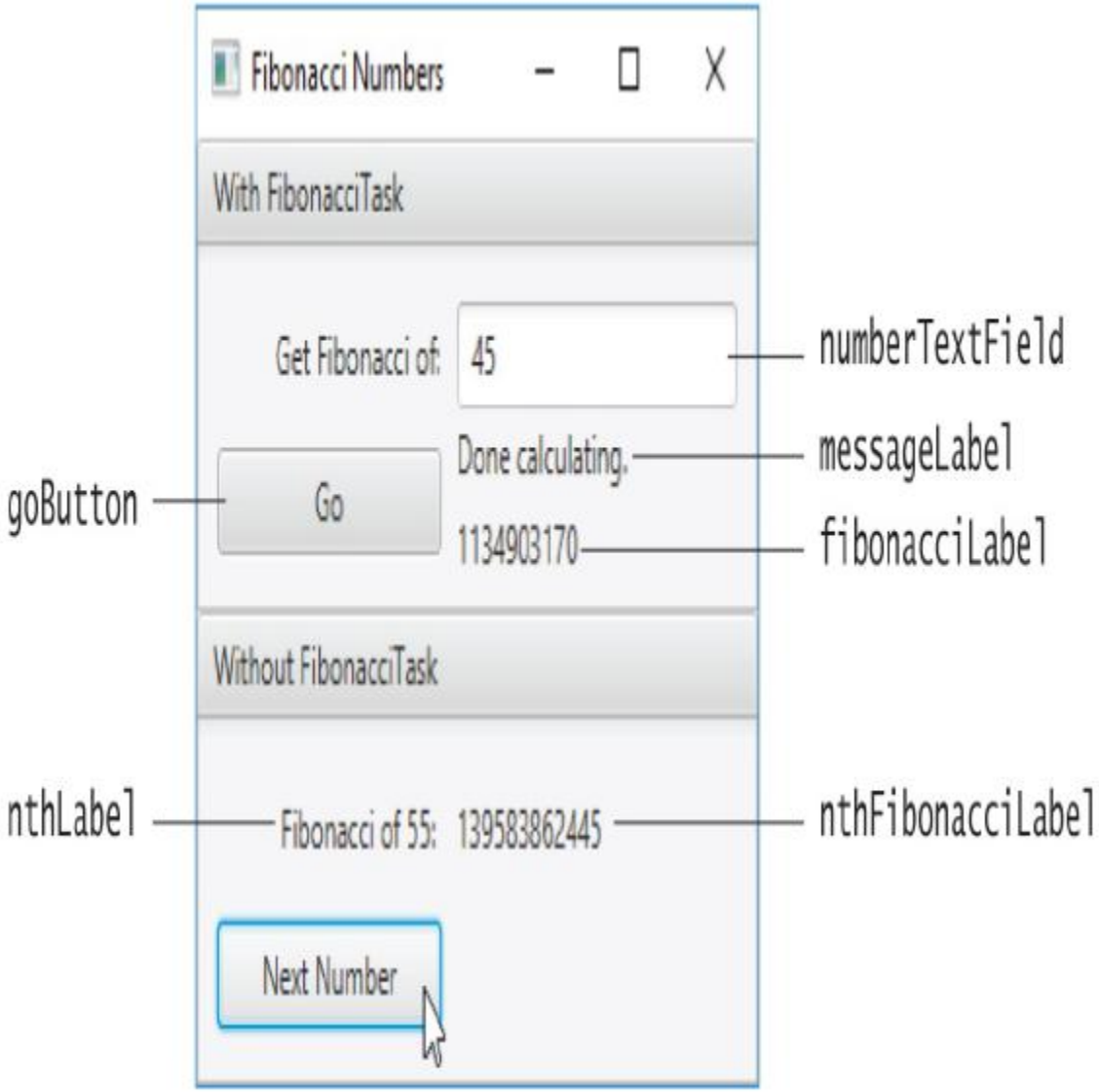
Fig. 26.22 | Concurrent collections summary (package `java.util.concurrent`).

23.11 Multithreading in JavaFX

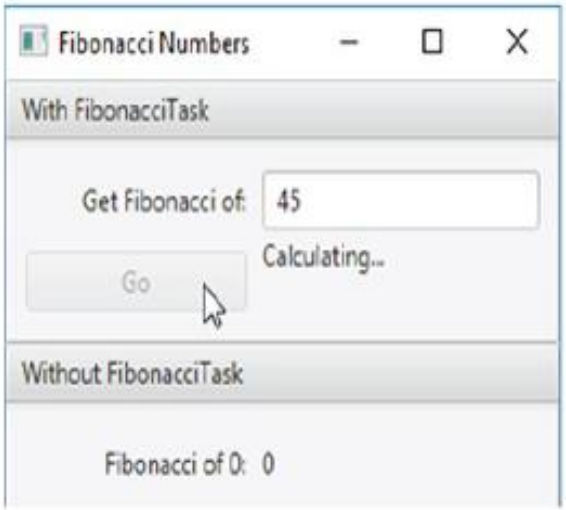
- All JavaFX applications have a single thread, called the **JavaFX application thread**, to handle interactions with the application's controls.
- All tasks that require interaction with an application's GUI are placed in an *event queue* and are executed sequentially by the JavaFX application thread.

23.11 Multithreading in JavaFX（cont.）

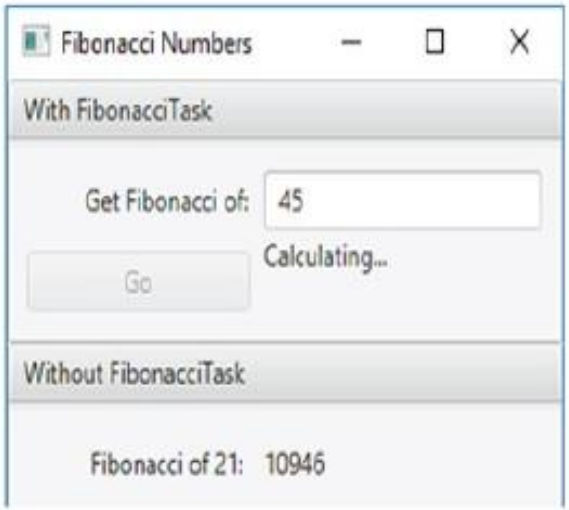
- *JavaFX's scene graph is not thread safe—its nodes cannot be manipulated by multiple threads without the risk of incorrect results that might corrupt the scene graph.*


Fibonacci Numbers
With FibonacciTask
Get Fibonacci of:
45
numberTextField
Done calculating.
messageLabel
goButton
Go
1134903170
fibonacciLabel
Without FibonacciTask
nthLabel
Fibonacci of 55:
139583862445
nthFibonacciLabel
Next Number

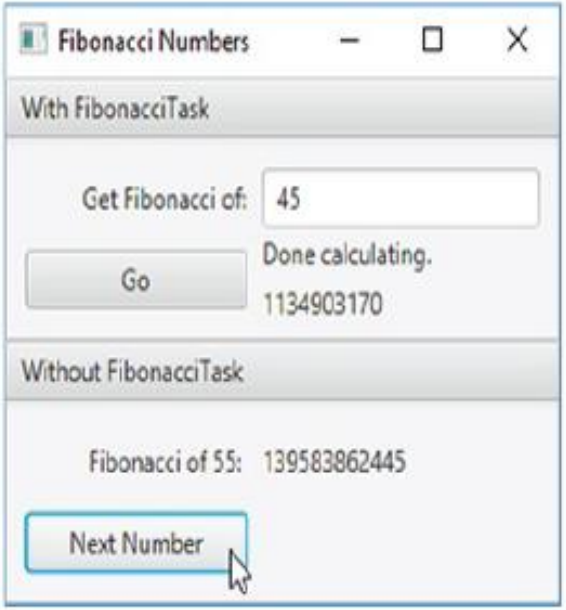
a) Begin calculating Fibonacci of 45 in the background



b) Calculating other Fibonacci values while Fibonacci of 45 continues calculating



c) Fibonacci of 45 calculation finishes



23.12 Interfaces `Callable` and `Future`

- The `Callable` interface (of package `java.util.concurrent`) declares a single method named `call`.

23.13 Java SE 7: Fork/Join Framework

- Java SE 7's concurrency APIs include the new fork/join framework, which helps programmers parallelize algorithms.
- The fork/join framework is well suited to divide-and-conquer-style algorithms, such as the merge sort in Section 23.3.3.
- The fork/join framework can be used to create parallel tasks so that they can be distributed across multiple processors and be truly performed in parallel—the details of assigning the parallel tasks to different processors are handled for you by the framework.

23.14 Java SE 8: Sequential vs. Parallel Streams

```
10  public class StreamStatisticsComparison
11  {
12     public static void main(String[] args)
13     {
14        SecureRandom random = new SecureRandom();
15
16        // create array of random long values
17        long[] values = random.longs(10_000_000, 1, 1001).toArray();
18
19        // perform calculcations separately
20        Instant separateStart = Instant.now();
21        long count = Arrays.stream(values).count();
22        long sum = Arrays.stream(values).sum();
23        long min = Arrays.stream(values).min().getAsLong();
24        long max = Arrays.stream(values).max().getAsLong();
25        double average = Arrays.stream(values).average().getAsDouble();
26        Instant separateEnd = Instant.now();
27
28        // display results
29        System.out.println("Calculations performed separately");
30        System.out.printf("    count: %,d%n", count);
31        System.out.printf("      sum: %,d%n", sum);
32        System.out.printf("      min: %,d%n", min);
33        System.out.printf("      max: %,d%n", max);
34        System.out.printf("  average: %f%n", average);
35        System.out.printf("Total time in milliseconds: %d%n%n",
36           Duration.between(separateStart, separateEnd).toMillis());
37
```

```
38          // time sum operation with sequential stream
39          LongStream stream1 = Arrays.stream(values);
40          System.out.println("Calculating statistics on sequential stream")
41          Instant sequentialStart = Instant.now();
42          LongSummaryStatistics results1 = stream1.summaryStatistics();
43          Instant sequentialEnd = Instant.now();
44
45          // display results
46          displayStatistics(results1);
47          System.out.printf("Total time in milliseconds: %d%n%n",
48             Duration.between(sequentialStart, sequentialEnd).toMillis());
49
50          // time sum operation with parallel stream
51          LongStream stream2 = Arrays.stream(values).parallel();
52          System.out.println("Calculating statistics on parallel stream");
53          Instant parallelStart = Instant.now();
54          LongSummaryStatistics results2 = stream2.summaryStatistics();
55          Instant parallelEnd = Instant.now();
56
57          // display results
58          displayStatistics(results1);
59          System.out.printf("Total time in milliseconds: %d%n%n",
60             Duration.between(parallelStart, parallelEnd).toMillis());
61       }
62
63       // display's LongSummaryStatistics values
64       private static void displayStatistics(LongSummaryStatistics stats)
65       {
66          System.out.println("Statistics");
67          System.out.printf("    count: %,d%n", stats.getCount());
68          System.out.printf("      sum: %,d%n", stats.getSum());
69          System.out.printf("      min: %,d%n", stats.getMin());
70          System.out.printf("      max: %,d%n", stats.getMax());
```

Swing component concurrency

```
Platform.runLater(new Runnable() {

      @Override public void run() {
            ……
        }
 });
```

```
Platform.runLater()->{……}}
```