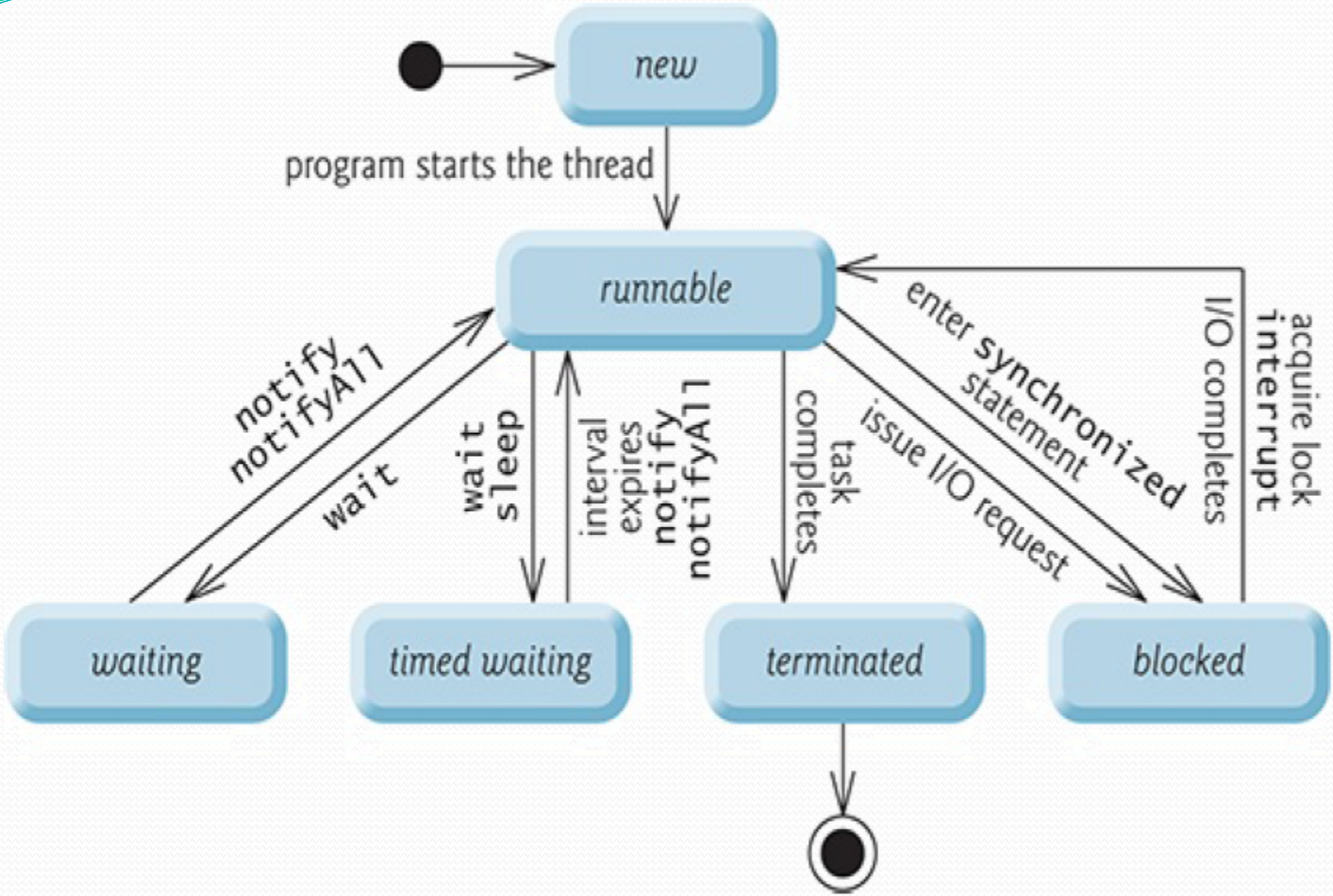
Concurrency

Chapter 23

Thread States and Life Cycle



Thread life-cycle UML state diagram.

New and Runnable States

- A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread, which places it in the runnable state. A thread in the runnable state is considered to be executing its task.

Waiting State

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

Timed Waiting State

- A runnable thread can enter the timed waiting state for a specified interval of time. It transitions back to the runnable state when that time interval expires or when the event it's waiting for occurs. Timed waiting threads and waiting threads cannot use a processor, even if one is available.

- Another way to place a thread
- in the *timed waiting* state is to put a *runnable* thread to sleep— 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.

Blocked State

- A runnable thread transitions to the blocked state when it attempts to perform a task that cannot be completed immediately and it must temporarily wait until that task completes. For example, when a thread issues an input/output request, the operating system blocks the thread from executing until that I/O request completes—at that point, the blocked
- thread transitions to the runnable state, so it can resume execution. A blocked thread cannot use a processor, even if one is available.

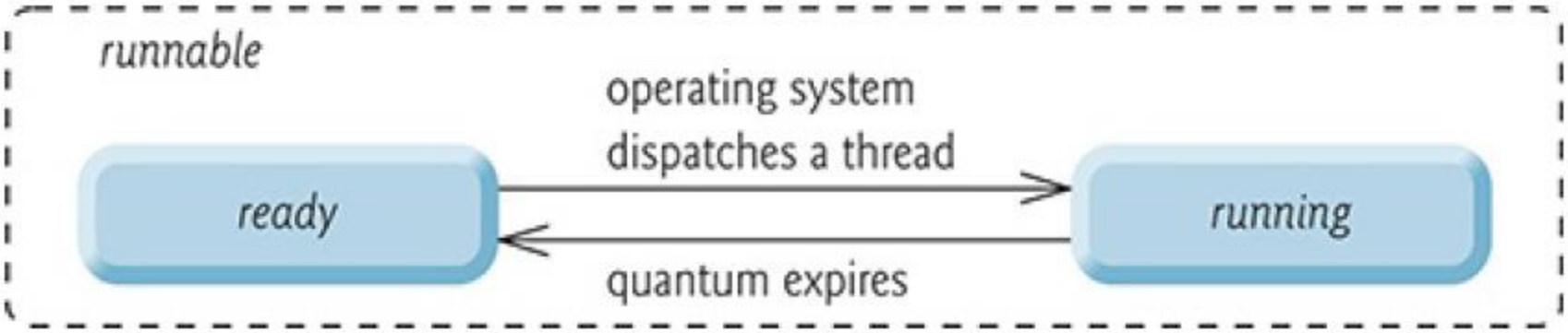
Terminated State

- A runnable thread enters the terminated state (sometimes called the dead state) when it successfully completes its task or otherwise terminates (perhaps due to an error). In the UML state diagram, the terminated state is followed by the UML final state to indicate the end of the state transitions.

Operating-SystemView of the Runnable State

- At the operating system level, Java's runnable state typically encompasses two separate states. The operating system hides these states from the JVM, which sees only the **runnable state**. When a thread first transitions to the runnable state from the new state, it's 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. In most operating systems, each thread is given a small amount of processor time—called a quantum or timeslice—with which to perform its task.

Operating-SystemView of the Runnable State



Thread Priorities and Thread Scheduling

- Thread priority that helps determine the order in which threads are scheduled. Each new thread inherits the priority of the thread that created it. Informally, higher-priority threads are more important to a program and should be allocated processor time before lower-priority threads. ***Nevertheless, thread priorities cannot guarantee the order in which threads execute.***

Thread scheduler

- An operating system's thread scheduler determines which thread runs next.

- One simple thread-scheduler implementation keeps the highest-priority thread running at all times and, if there's more than one highest-priority thread, ensures that all such threads execute for a quantum each in **round-robin** fashion. This process continues until all threads run to completion.

Executor *interface*

- *It's recommended that you do not explicitly create and use* Threads *to implement concurrency, but rather use the* Executor *interface (described in Section 23.3).* The
- Thread class does contain some useful static methods,
- which you *will* use later in the chapter.

Portability

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

Indefinite Postponement and Deadlock

- When a higher-priority thread enters the ready state, the operating system generally preempts the running thread (an operation known as preemptive scheduling). Depending on the operating system, a steady influx of higher-priority threads could postpone—possibly indefinitely—the execution of lower-priority threads. Such indefinite postponement is sometimes referred to more colorfully as **starvation.**
- Operating systems employ a technique called **aging** to prevent starvation—as a thread waits in the ready state, the operating system gradually increases the thread's priority to ensure that the thread will eventually run.

Indefinite Postponement and Deadlock

- Another problem related to indefinite postponement is called **deadlock**. This 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.

Creating and Executing Threads with the Executor Framework

This section demonstrates how to perform concurrent tasks in an application by using Executors and Runnable objects.

Creating Concurrent Tasks with the Runnable Interface

- You implement the Runnable interface (of package java.lang) to specify 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.

Runnable Interface

```
public class RunnableDemo {
    public static void main(String[] args)
    {
        System.out.println("Main thread is- "
                           + Thread.currentThread().getName());
        Thread t1 = new Thread(new RunnableDemo().new RunnableImpl());
        t1.start();
    }
    private class RunnableImpl implements Runnable {
        public void run()
        {
            System.out.println(Thread.currentThread().getName()
                               + ", executing run() method!");
        }
    }
}
```

```
Output:
Main thread is- main
Thread-0, executing run() method!
```

create Threads by Extending Thread Class

```
import java.io.*;
import java.util.*;

public class AlphaPeeler extends Thread {
      // initiated run method for Thread
    public void run()
    {
        System.out.println("Thread Started Running...");
    }
    public static void main(String[] args)
    {
    AlphaPeeler g1 = new AlphaPeeler();
          // invoking Thread
        g1.run();
    }
}
```

Output:
Thread Started Running...

- See further examples:

- https://www.geeksforgeeks.org/java-threads/

Executing Runnable Objects with an Executor

Using Class Executors to Obtain an ExecutorService

- The ExecutorService interface (of package java.util.concurrent) extends Executor and declares various methods for managing the life cycle of an Executor. You obtain an ExecutorService object by calling one of the static methods declared in class Executors (of package java.util.concurrent). We use interface ExecutorService and a method of class Executors in our example (Fig. 23.4), which executes three tasks.

Using the ExecutorService to Manage Threads That Execute PrintTasks

```java
import java.security.SecureRandom;
public class PrintTask implements Runnable {
   private static final SecureRandom generator = new SecureRandom();
   private final int sleepTime; // random sleep time for thread
   private final String taskName;
   // constructor
   public PrintTask(String taskName) {
      this.taskName = taskName;

      // pick random sleep time between 0 and 5 seconds
      sleepTime = generator.nextInt(5000); // milliseconds
   }
   // method run contains the code that a thread will execute
   @Override
   public void run() {
      try { // put thread to sleep for sleepTime amount of time
         System.out.printf("%s going to sleep for %d milliseconds.%n",
            taskName, sleepTime);
         Thread.sleep(sleepTime); // put thread to sleep
      }
      catch (InterruptedException exception) {
         exception.printStackTrace();
         Thread.currentThread().interrupt(); // re-interrupt the thread
      }
      // print task name
      System.out.printf("%s done sleeping%n", taskName);
   }
}
```

```java
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
public class TaskExecutor {
   public static void main(String[] args) {
      // create and name each runnable
      PrintTask task1 = new PrintTask("task1");
      PrintTask task2 = new PrintTask("task2");
      PrintTask task3 = new PrintTask("task3");

      System.out.println("Starting Executor");

      // create ExecutorService to manage threads
      ExecutorService executorService = Executors.newCachedThreadPool();

      // start the three PrintTasks
      executorService.execute(task1); // start task1
      executorService.execute(task2); // start task2
      executorService.execute(task3); // start task3

      // shut down ExecutorService--it decides when to shut down threads
      executorService.shutdown();

      System.out.printf("Tasks started, main ends.%n%n");
   }
}
```

Output 1st run:

```
Starting Executor
task1 going to sleep for 3472 milliseconds.
task3 going to sleep for 4262 milliseconds.
Tasks started, main ends.

task2 going to sleep for 3306 milliseconds.
task2 done sleeping
task1 done sleeping
task3 done sleeping
```

Output 2nd Run:

```
Starting Executor
Tasks started, main ends.

task3 going to sleep for 2622 milliseconds.
task2 going to sleep for 1508 milliseconds.
task1 going to sleep for 1173 milliseconds.
task1 done sleeping
task2 done sleeping
task3 done sleeping
```

Thread Synchronization

Problems with shared object

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

thread synchronization

- The problem can be solved by giving only one thread at a time exclusive access to code that accesses the shared object. During that time, other threads desiring to access the object are kept waiting. When the thread with exclusive access finishes accessing the object, one of the waiting threads is allowed to proceed. This process, called thread synchronization

Immutable Data

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

Immutable Data

- 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.

Monitors

- A common way to perform synchronization is to use Java's built-in monitors. Every object has a monitor and a monitor lock (or intrinsic lock). The monitor ensures that its object's monitor lock is held by a maximum of only one thread at any time. Monitors and monitor locks can thus be used to enforce mutual exclusion.

Java Monitor Pattern

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

Unsynchronized Mutable Data Sharing

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

Class SimpleArray

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

- Next sample code: Class that manages an integer array to be shared by multiple threads. (*Caution:* The example of Figs. 23.5–23.7 is *not* thread safe.)

Class SimpleArray

```
import java.security.SecureRandom;
import java.util.Arrays;
public class SimpleArray { // CAUTION: NOT THREAD SAFE!
   private static final SecureRandom generator = new SecureRandom();
   private final int[] array; // the shared integer array
   private int writeIndex = 0; // shared index of next element to write
   // construct a SimpleArray of a given size
   public SimpleArray(int size) {array = new int[size];}
   public void add(int value) {// add a value to the shared array
      int position = writeIndex; // store the write index
      try {
         Thread.sleep(generator.nextInt(500));//put thread to sleep for 0-499 milliseconds
      }
      catch (InterruptedException ex) {
         Thread.currentThread().interrupt(); // re-interrupt the thread
      }
      array[position] = value; // put value in the appropriate element
      System.out.printf("%s wrote %2d to element %d.%n",
         Thread.currentThread().getName(), value, position);
      ++writeIndex; // increment index of element to be written next
      System.out.printf("Next write index: %d%n", writeIndex);
   }
   // used for outputting the contents of the shared integer array
   @Override
   public String toString() {
      return Arrays.toString(array);
   }
}
```

Class SimpleArray

```
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;
   }

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

Class SimpleArray

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;
public class SharedArrayTest {
   public static void main(String[] arg) {
      // construct the shared object
      SimpleArray sharedSimpleArray = new SimpleArray(6);
      // create two tasks to write to the shared SimpleArray
      ArrayWriter writer1 = new ArrayWriter(1, sharedSimpleArray);
      ArrayWriter writer2 = new ArrayWriter(11, sharedSimpleArray);
      // execute the tasks with an ExecutorService
      ExecutorService executorService = Executors.newCachedThreadPool();
      executorService.execute(writer1);
      executorService.execute(writer2);
      executorService.shutdown();
      try {
         // wait 1 minute for both writers to finish executing
         boolean tasksEnded =
            executorService.awaitTermination(1, TimeUnit.MINUTES);
         "Timed out while waiting for tasks to finish.");
         }
      }
      catch (InterruptedException ex) {
         ex.printStackTrace();
      }
   }
}
```

Class SimpleArray

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

Contents of SimpleArray:
[1, 12, 2, 3, 13, 0]
```

Analysis on output

- The value 1 was written to element 0, then overwritten later by the value 11. Also, when writeIndex was incremented to 3, nothing was written to that element, as indicated by the 0 in that element of the array.

- We call Thread method sleep between operations on the shared mutable data to emphasize the unpredictability of thread scheduling and to increase the likelihood of producing erroneous output. Even if these operations were allowed to proceed at their normal pace, you could still see errors in the program's output.

Synchronized Mutable Data Sharing—Making Operations Atomic

Software Engineering Observation

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

Mutable Data Sharing

```
import java.security.SecureRandom;
import java.util.Arrays;
public class SimpleArray {
   private static final SecureRandom generator = new SecureRandom();
   private final int[] array; // the shared integer array
   private int writeIndex = 0; // index of next element to be written
   // construct a SimpleArray of a given size
   public SimpleArray(int size) {array = new int[size];}
   // add a value to the shared array
   public synchronized void add(int value) {
      int position = writeIndex; // store the write index
      try { // in real applications, you shouldn't sleep while holding a lock
         Thread.sleep(generator.nextInt(500)); // for demo only
      }
      catch (InterruptedException ex) {
         Thread.currentThread().interrupt();
      }
      array[position] = value; // put value in the appropriate element
      System.out.printf("%s wrote %2d to element %d.%n",
         Thread.currentThread().getName(), value, position);
      ++writeIndex; // increment index of element to be written next
      System.out.printf("Next write index: %d%n", writeIndex);
   }
   // used for outputting the contents of the shared integer array
   @Override
   public synchronized String toString() {
      return Arrays.toString(array);
   }
}
```

Class that manages an integer array to be shared by multiple threads with synchronization.

Output (old):	Output(new):
pool-1-thread-2 wrote 11 to element 0. Next write index: 1 pool-1-thread-1 wrote 1 to element 0. Next write index: 2 pool-1-thread-1 wrote 2 to element 2. Next write index: 3 pool-1-thread-2 wrote 12 to element 1. Next write index: 4 pool-1-thread-2 wrote 13 to element 4. Next write index: 5 pool-1-thread-1 wrote 3 to element 3. Next write index: 6 Contents of SimpleArray: [1, 12, 2, 3, 13, 0]	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-2 wrote 11 to element 2. Next write index: 3 pool-1-thread-2 wrote 12 to element 3. Next write index: 4 pool-1-thread-2 wrote 13 to element 4. Next write index: 5 pool-1-thread-1 wrote 3 to element 5. Next write index: 6 Contents of SimpleArray: [1, 2, 11, 12, 13, 3]

Performance Tip

- 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.

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. The producer/consumer relationship separates the task of identifying work to be done from the tasks involved in actually carrying out the work.

Producer/Consumer Examples

- Example 1: Print spooling. Although a printer might not be available when you want to print from an application (i.e., the producer), you can still "complete" the print task, as the data is temporarily placed on disk until the printer becomes available. Similarly, when the printer (i.e., a consumer) is available, it doesn't have to wait until a current user wants to print. The spooled print jobs can be printed as soon as the printer becomes available.
- Example 2: Application that copies data onto DVDs by placing data in a fixed-size buffer, which is emptied as the DVD drive "burns" the data onto the DVD.

- Our next example (Figs. 23.9–23.13) implements a producer/consumer relationship without the proper synchronization. A producer thread writes the numbers 1 through 10 into a shared buffer—a single memory location shared between two threads (a single int variable called buffer in line 5 of Fig. 23.12 in this example).
- The consumer thread reads this data from the shared buffer and displays the data. The program's output shows the values that the producer writes (produces) into the shared buffer and the values that the consumer reads (consumes) from the shared buffer.
- Data can be lost or garbled if the producer places new data into the shared buffer before the consumer reads the previous data.

UnsynchronizedBuffer

```
public interface Buffer {
   // place int value into Buffer
   public void blockingPut(int value) throws InterruptedException;

   // return int value from Buffer
   public int blockingGet() throws InterruptedException;
}
```

```
public class UnsynchronizedBuffer implements Buffer {
   private int buffer = -1; // shared by producer and consumer threads

   // place value into buffer
   @Override
   public void blockingPut(int value) throws InterruptedException {
      System.out.printf("Producer writes\t%2d", value);
      buffer = value;
   }

   // return value from buffer
   @Override
   public int blockingGet() throws InterruptedException {
      System.out.printf("Consumer reads\t%2d", buffer);
      return buffer;
   }
}
```

Consumer class

```
import java.security.SecureRandom;
public class Consumer implements Runnable {
   private static final SecureRandom generator = new SecureRandom();
   private final Buffer sharedLocation; // reference to shared object
   public Consumer(Buffer sharedLocation) { // constructor
      this.sharedLocation = sharedLocation;
   }
   // read sharedLocation's value 10 times and sum the values
   @Override
   public void run() {
      int sum = 0;
      for (int count = 1; count <= 10; count++) {
         // sleep 0 to 3 seconds, read value from buffer and add to sum
         try {
            Thread.sleep(generator.nextInt(3000));
            sum += sharedLocation.blockingGet();
            System.out.printf("\t\t\t%2d%n", sum);
         }
         catch (InterruptedException exception) {
            Thread.currentThread().interrupt();
         }
      }
      System.out.printf("%n%s %d%n%s%n",
         "Consumer read values totaling", sum, "Terminating Consumer");
   }
}
```

Producer class

```
import java.security.SecureRandom;
public class Producer implements Runnable {
   private static final SecureRandom generator = new SecureRandom();
   private final Buffer sharedLocation; // reference to shared object
   public Producer(Buffer sharedLocation) { // constructor
      this.sharedLocation = sharedLocation;
   }
   // store values from 1 to 10 in sharedLocation
   @Override
   public void run() {
      int sum = 0;
      for (int count = 1; count <= 10; count++) {
         try { // sleep 0 to 3 seconds, then place value in Buffer
            Thread.sleep(generator.nextInt(3000)); // random sleep
            sharedLocation.blockingPut(count); // set value in buffer
            sum += count; // increment sum of values
            System.out.printf("\t%2d%n", sum);
         }
         catch (InterruptedException exception) {
            Thread.currentThread().interrupt();
         }
      }
      System.out.printf(
         "Producer done producing%nTerminating Producer%n");
   }
}
```

Driver code

```
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
      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);
   }
}
```

Output:

```
ActionValueSum of ProducedSum of Consumed
-----------------------------------------

Consumer reads-1-1
Consumer reads-1-2
Producer writes 1 1
Consumer reads 1-1
Producer writes 2 3
Consumer reads 2 1
Producer writes 3 6
Producer writes 410
Consumer reads 4 5
Producer writes 515
Producer writes 621
Consumer reads 611
Consumer reads 617
Producer writes 728
Consumer reads 724
Producer writes 836
Consumer reads 832
Consumer reads 840

Consumer read values totaling 40
Terminating Consumer
Producer writes 945
Producer writes1055
Producer done producing
Terminating Producer
```

Output Analysis

- interface Buffer (line 4), but does *not* synchronize access to the buffer's state—we purposely do this to demonstrate the problems that occur when multiple threads access *shared* mutable data *without* synchronization. Line 5 declares instance variable buffer and initializes it to –1. This value is used to demonstrate the case in which the Consumer attempts to consume a value before the Producer ever places a value in buffer. Again, methods blockingPut (lines 8–12) and blockingGet (lines 15–19) do not synchronize access to the buffer instance variable. Method blockingPut simply assigns its argument to buffer (line 11), and method blockingGet simply returns the value of buffer (line 18). As you'll see in Fig. 23.13, UnsynchronizedBuffer object is shared between the Producer and the Consumer.

Action	Value	Sum of Produced	Sum of Consumed	
writes				
Producer writes	2	3		—*1 lost*
Producer writes	3	6		—*2 lost*
Consumer reads	3		3	
Producer writes	4	10		
Consumer reads	4		7	
Producer writes	5	15		
Producer writes	6	21		—*5 lost*
Producer writes	7	28		—*6 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 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
```

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 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	— *9 never read*

```
Producer done producing
Terminating Producer
```

Error-Prevention Tip

- Access to a shared object by concurrent threads must be controlled carefully or a program may produce incorrect results.
- To solve the problems of lost and duplicated data, Section 23.6 presents an example in which we use an ArrayBlockingQueue (from package java.util.concurrent) to synchronize access to the shared object, guaranteeing that each and every value will be processed once and only once.

Producer/Consumer Relationship: ArrayBlockingQueue

Creating a synchronized buffer using an ArrayBlockingQueue.

```
public interface Buffer {
   // place int value into Buffer
   public void blockingPut(int value) throws
InterruptedException;

   // return int value from Buffer
   public int blockingGet() throws InterruptedException;
}
```

```
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
   @Override
   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
   @Override
   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;
   } }
```

Producer class

No change in code
Same as previous producer.

```
import java.security.SecureRandom;

public class Producer implements Runnable {
   private static final SecureRandom generator = new SecureRandom();
   private final Buffer sharedLocation; // reference to shared object
   public Producer(Buffer sharedLocation) {// constructor
      this.sharedLocation = sharedLocation;
   }
   // store values from 1 to 10 in sharedLocation
   @Override
   public void run() {
      int sum = 0;
      for (int count = 1; count <= 10; count++) {
         try { // sleep 0 to 3 seconds, then place value in Buffer
            Thread.sleep(generator.nextInt(3000)); // random sleep
            sharedLocation.blockingPut(count); // set value in buffer
            sum += count; // increment sum of values
         }
         catch (InterruptedException exception) {
            Thread.currentThread().interrupt();
         }
      }
      System.out.printf(
         "Producer done producing%nTerminating Producer%n");
   }
}
```

Consumer class

No change in code
Same as previous consumer.

```
import java.security.SecureRandom;
public class Consumer implements Runnable {
   private static final SecureRandom generator = new SecureRandom();
   private final Buffer sharedLocation; // reference to shared object
   public Consumer(Buffer sharedLocation) {// constructor
      this.sharedLocation = sharedLocation;
   }
   // read sharedLocation's value 10 times and sum the values
   @Override
   public void run() {
      int sum = 0;
      for (int count = 1; count <= 10; count++) {
         // sleep 0 to 3 seconds, read value from buffer and add to sum
         try {
            Thread.sleep(generator.nextInt(3000));
            sum += sharedLocation.blockingGet();
         }
         catch (InterruptedException exception) {
            Thread.currentThread().interrupt();
         }
      }
      System.out.printf("%n%s %d%n%s%n",
         "Consumer read values totaling", sum, "Terminating Consumer");
   }
}
```

Output:

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

Consumer read values totaling 55
Terminating Consumer
```