CNT 4714: Enterprise Computing Spring 2012

Programming Multithreaded Applications in Java Part 2

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cnt4714/spr2012

Department of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida

Threads

- In the previous section of notes the thread examples all involved threads which were unsynchronized. None of the threads actually needed to communicate with one another and they did not require access to a shared object.
- The threads we've seen so far fall into the category of unrelated threads. These are threads which do different tasks and do not interact with one another.
- A slightly more complex form of threading involves threads which are related but unsynchronized. In this case, multiple threads operate on different pieces of the same data structure. An example of this type of threading is illustrated on the next page with a threaded program to determine if a number is prime.

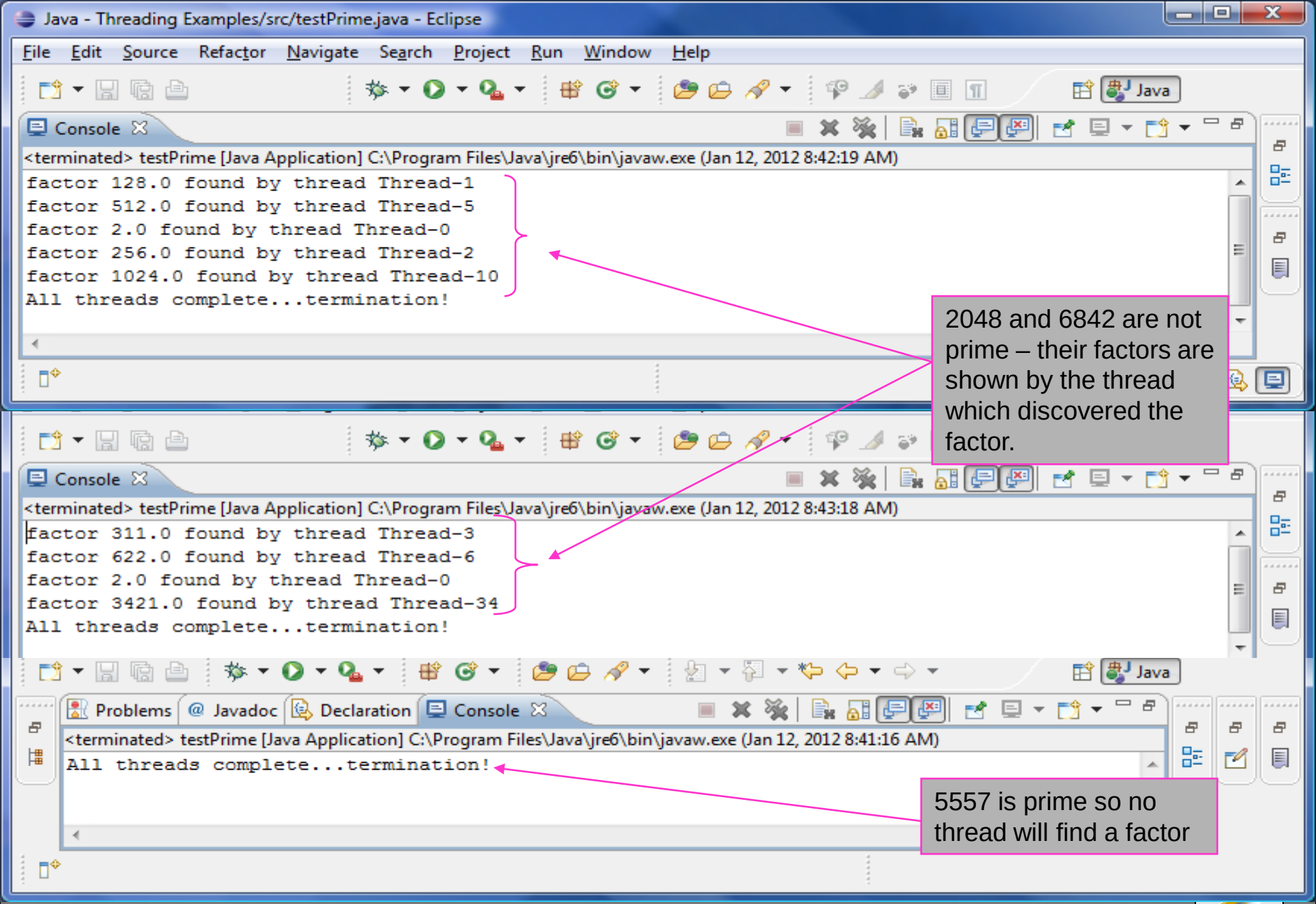
Prime Number Tester Class

```
//class for threaded prime number testing
//no inheritance issues so using the simple form of thread creation
class testRange extends Thread {
   static long possPrime;
    long from, to; //test range for a thread
     //constructor
    //record the number to be tested and the range to be tried
    testRange(int argFrom, long argpossPrime) {
           possPrime = argpossPrime;
           if (argFrom ==0) from = 2; else from = argFrom;
           to=argFrom+99;
    }
    //implementation of run
    public void run() {
       for (long i=from; i <= to && i<possPrime; i++) {
            if (possPrime % i == 0) {
                //i divides possPrime exactly
               System.out.println("factor " + i + " found by thread " + getName() );
               break;  //exit for loop immediately
            }
            yield(); //suspend thread
       }
   }
}
```


Driver Class for Prime Number Tester

```
//driver class to demonstrate threaded prime number tester
public class testPrime {
  public static void main (String s[]) {
            //number to be tested for primality is entered as a command line argument
            //examples: 5557 is prime, 6841 is prime, 6842 is not prime
            long possPrime = Long.parseLong(s[0]);
                    int centuries = (int) (possPrime/100) + 1;
                    for (int i=0; i<centuries;i++) {
                        new testRange(i*100, possPrime).start();
                    }
          }
}
```

- This is an example of related but unsynchronized threads. In this case the threads are related since they are each working on a piece of the same data, but approach it from a slightly different perspective. However, they are unsynchronized since they do not share information.


Java - Threading Examples/src/testPrime.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Console
<terminated> testPrime [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 12, 2012 8:42:19 AM)
factor 128.0 found by thread Thread-1
factor 512.0 found by thread Thread-5
factor 2.0 found by thread Thread-0
factor 256.0 found by thread Thread-2
factor 1024.0 found by thread Thread-10
All threads complete...termination!
2048 and 6842 are not prime – their factors are shown by the thread which discovered the factor.
Console
<terminated> testPrime [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 12, 2012 8:43:18 AM)
factor 311.0 found by thread Thread-3
factor 622.0 found by thread Thread-6
factor 2.0 found by thread Thread-0
factor 3421.0 found by thread Thread-34
All threads complete...termination!
Problems Javadoc Declaration Console
<terminated> testPrime [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 12, 2012 8:41:16 AM)
All threads complete...termination!
5557 is prime so no thread will find a factor

Related and Synchronized Threads

- The most complicated type of threaded application involves threads which interact with each other. These are related synchronized threads (also referred to as cooperating threads).
- Without synchronization when multiple threads share an object and that object is modified by one or more of the threads, indeterminate results may occur. This is known as a data race or race condition.
- The following example illustrates a race condition. In this example, we simulate a steam boiler and the reading of its pressure. The program starts 10 unsynchronized threads which each read the pressure of the boiler and if it is found to be below the safe limit, the pressure in the boiler is increased by 15psi. If the pressure is found to already be above the safe limit, the pressure is not increased. Looking at the results you can clearly see the problem with this approach.

SteamBoiler1.java | pressure1.java

Class to Simulate a Steam Boiler – Pressure Gauge

```
// class to simulate a steam boiler to illustrate a race condition
//in unsynchronized threads
public class SteamBoiler1 {
   static int pressureGauge = 0;
    static final int safetyLimit = 50;

    public static void main(String [] args) {
      pressure1 []psi = new pressure1[10];
      for (int i = 0; i < 10; i++) {
         psi[i] = new pressure1();
         psi[i].start();
      }
      //we now have 10 threads in execution to monitor the pressure
      try {
         for (int i = 0; i < 10; i++)
             psi[i].join();  //wait for the thread to finish
             //psi[i].sleep(200);
        }
        catch (Exception e) { } //do nothing
        System.out.println();
        System.out.print("Gauge reads " + pressureGauge + ", the safe limit is " + safetyLimit);
        if (pressureGauge > safetyLimit)
            System.out.println("...B O O M ! ! !");
        else System.out.println("...System OK!");
    }
}
```


SteamBoiler1.java | pressure1.java

```
//thread class to raise the pressure in the Boiler
class pressure1 extends Thread {
    void RaisePressure() {
       if (SteamBoiler1.pressureGauge < SteamBoiler1.safetyLimit-15) {
          //wait briefly to simulate some calculations
          try {sleep(100); } catch (Exception e) { }
          SteamBoiler1.pressureGauge+= 15; //raise the pressure 15 psi
          System.out.println("Thread " + this.getName() + " finds pressure within limits - increases pr
       }
       else
          System.out.println("Thread" + this.getName() + " finds pressure too high - do nothing");
    }
  public void run() {
      RaisePressure();  //this thread is to raise the pressure
  }
}
```

Thread Class to Read Steam Boiler Pressure Gauge and Increase the Pressure if Within Range

SteamBoiler1.java | pressure1.java

```
//thread class to raise the pressure in the Boiler
class pressure1 extends Thread {
    void RaisePressure() {
       if (SteamBoiler1.pressureGauge < SteamBoiler1.safetyLimit-15) {
          //wait briefly to simulate some calculations
          try {sleep(100); } catch (Exception e) { }
          SteamBoiler1.pressureGauge+= 15; //raise the pressure 15 psi
          System.out.println("Thread " + this.getName() + " finds pressure within limits - increases pr
       }
       else
```

This is what caused the race condition to occur.

Console

```
<terminated> SteamBoiler1 [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 12, 2012 10:21:34 AM)
Thread Thread-2 finds pressure within limits - increases pressure
Thread Thread-4 finds pressure within limits - increases pressure
Thread Thread-3 finds pressure within limits - increases pressure
Thread Thread-1 finds pressure within limits - increases pressure
Thread Thread-7 finds pressure within limits - increases pressure
Thread Thread-5 finds pressure within limits - increases pressure
Thread Thread-8 finds pressure within limits - increases pressure
Thread Thread-6 finds pressure within limits - increases pressure
Thread Thread-9 finds pressure within limits - increases pressure

Gauge reads 120, the safe limit is 50...B O O M ! ! !
```

Output From Execution Illustrating the Race Condition

SteamBoiler1.java | pressure1.java

```
//thread class to raise the pressure in the Boiler
class pressure1 extends Thread {
    void RaisePressure() {
       if (SteamBoiler1.pressureGauge < SteamBoiler1.safetyLimit-15) {
          //wait briefly to simulate some calculations
         // try {sleep(100); } catch (Exception e) { }
          SteamBoiler1.pressureGauge+= 15; //raise the pressure 15 psi
          System.out.println("Thread " + this.getName() + " finds pressure within limits - increases pr
       }
       else
```

See. . ., I told you so!

Console

<terminated> SteamBoiler1 [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 12, 2012 10:25:21 AM)

```
ThreadThread-4 finds pressure too high - do nothing
ThreadThread-3 finds pressure too high - do nothing
Thread Thread-2 finds pressure within limits - increases pressure
Thread Thread-1 finds pressure within limits - increases pressure
ThreadThread-5 finds pressure too high - do nothing
ThreadThread-6 finds pressure too high - do nothing
ThreadThread-7 finds pressure too high - do nothing
ThreadThread-8 finds pressure too high - do nothing
ThreadThread-9 finds pressure too high - do nothing

Gauge reads 45, the safe limit is 50...System OK!
```

Output From Execution Illustrating No Race Condition

Interesting Note on Race Conditions

- You may remember the large North American power blackout that occurred on August 14, 2003. Roughly 50 million people lost electrical power in a region stretching from Michigan through Canada to New York City. It took three days to restore service to some areas.
- See http://en.wikipedia.org/wiki/Northeast_blackout_of_2003 (scroll down to computer failure)
- There were several factors that contributed to the blackout, but the official report highlights the failure of the alarm monitoring software which was written in C++ by GE Energy. The software failure wrongly led operators to believe that all was well, and precluded them from rebalancing the power load before the blackout cascaded out of control.
- Because the consequences of the software failure were so severe, the bug was analyzed exhaustively. The root cause was finally identified by artificially introducing delays in the code (just like we did in the previous example). There were two threads that wrote to a common data structure, and through a coding error, they could both update it simultaneously. It was a classic race condition, and eventually the program "lost the race", leaving the structure in an inconsistent state. That in turn caused the alarm event handler to spin in an infinite loop, instead of raising the alarm. The largest power failure in the history of the US and Canada was caused by a race condition bug in some threaded C++ code. Java is equally vulnerable to this kind of bug.

The Therac-25 Accidents

- Starting in 1976, the Therac-25 treatment system, built by Atomic Energy of Canada Limited (AECL) and COR MeV of France, was used to fight cancer by providing radiation to a specific part of the body in the hope of destroying tumors.
- See http://en.wikipedia.org/wiki/Therac-25 (See last line of the Problem description.)
- Six known Therac-25 accidents have been documented, all involved massive overdoses (100x normal dose) of radiation and three resulted in the death of the patient, serious long-term injury and disfigurement occurred in the other cases. Patients received an estimated 17,000 to 25,000 rads to very small body areas. By comparison, doses of 1000 rads can be fatal if delivered to the whole body.
- Analysis determined that the primary cause of the overdoses was faulty software. The software was written in assembly language and was developed and tested by the same person. The software included a scheduler and concurrency in its design. When the system was first built, operators complained that it took too long to enter the treatment plan into the computer. As a result, the software was modified to allow operators to quickly enter treatment data by simply pressing the Enter key when an input value did not require changing.

The Therac-25 Accidents (cont.)

- This modification created a synchronization error (a race condition developed) between the code that read the data entered by the operator and the code controlling the machine. As a result, the actions of the machine would lag behind the commands the operator entered. The machine appeared to administer the dose entered by the operator, but it fact had an improper setting that focused radiation at full power to a tiny spot on the body.
- The race condition was subsequently found to occur only when a certain non-typical keystroke sequence was entered (an "X" to select a 25MeV photon followed by "cursor-up" ,"E" to correctly set the 25MeV Electron mode, then "Enter"), since this sequence of keystrokes did not occur very often, the error went unnoticed for a long time.
- AECL was ultimately cited for improperly testing the software, which was only tested on site in hospitals after a machine was assembled in place.
- The designer had reused software from older Therac-6 and Therac-20 models that had hardware interlocks which masked the software defects. Some operators noted that certain situations caused the machines to display MALFUNCTION followed by a number between 1 and 64 on the display screen. However, the user manual did not explain nor even address error codes, so the operators pressed the "P" key (for proceed), to override the warning and proceed with the treatment.

Thread Synchronization

- To prevent a race condition, access to the shared object must be properly synchronized.
  - Lost update problem: one thread is in the process of updating the shared value and another thread also attempts to update the value.
  - Even worse is when only part of the object is updated by each thread in which case part of the object reflects information from one thread while another part of the same object reflects information from another thread.
- The problem can be solved by giving one thread at a time exclusive access to code that manipulates the shared object. During that time, other threads desiring to manipulate the object must be forced to wait.

Thread Synchronization (cont.)

- When the thread with exclusive access to the object finishes manipulating the object, one of the blocked threads will be allowed to proceed and access the shared object.
  - The next selected thread will be based on some protocol. The most common of these is simply FCFS (priority-queue based).
- In this fashion, each thread accessing the shared object excludes all other threads from accessing the object simultaneously. This is the process known as mutual exclusion.
- Mutual exclusion allows the programmer to perform thread synchronization, which coordinates access to shared objects by concurrent threads.

Synchronization Techniques

- There have been many different methods used to synchronize concurrent processes. Some of the more common ones are:
  - Test and Set Instructions. All general purpose processors now have this kind of instruction, and it is used to build higher-level synchronization constructs. Test and set does not block, that must be built on top of it.
  - *p* and *v* semaphores. Introduced by Dijkstra in the 1960's and was the main synchronization primitive for a long time. Its easy to build semaphores from test and set instructions. Semaphores are low-level and can be hard for programmers to read and debug. For your information the *p* is short for the Dutch words *proberen te verlangen* which means to "try to decrement" and the *v* stands for *verhogen* which means to increment.

Synchronization Techniques (cont.)

- Read/write Locks. These are also commonly referred to as mutexes (although some people still use the term `mutex` to refer to a semaphore.) A lock provides a simple ”turnstile”: only one thread at a time can be going through (executing in) a block protected by a lock. Again, it is easy to build a lock from semaphores.

- Monitors. A monitor is a higher-level synchronization construct built out of a lock plus a variable that keeps track of some related condition, such as “the number of unconsumed bytes in the buffer”. It is easy to build monitors from read/write locks. A monitor defines several methods as a part of its protocol. Two of those predefined methods are `wait()` and `notify()`.

Types of Synchronization

- There are two basic types of synchronization between threads:

  1. Mutual exclusion is used to protect certain critical sections of code from being executed simultaneously by two or more threads. (Synchronization without cooperation.)

  2. Signal-wait is used when one thread need to wait until another thread has completed some action before continuing. (Synchronization with cooperation.)

- Java includes mechanisms for both types of synchronization.

- All synchronization in Java is built around locks. Every Java object has an associated lock. Using appropriate syntax, you can specify that the lock for an object be locked when a method is invoked. Any further attempts to call a method for the locked object by other threads cause those threads to be blocked until the lock is unlocked.

Thread Synchronization In Java

- Any object can contain an object that implements the `Lock` interface (package `java.util.concurrent.locks`).
- A thread calls the `Lock`'s `lock()` method to obtain 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 thread releases the lock (by invoking the `Lock`'s `unlock()` method).
- If there are several threads trying to invoke method `lock()` on the same `Lock` object, only one thread may obtain the lock, with all other threads being placed into the wait state.

An Aside on Reentrant Locks

- Class `ReentrantLock` (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 this is set to true, the ReentrantLock's fairness policy states that the longest-waiting thread will acquire the lock when it is available. If set to false, there is no guarantee as to which waiting thread will acquire the lock when it becomes available.
- Using a lock with a fairness policy helps avoid indefinite postponement (starvation) but can also dramatically reduce the overall efficiency of a program. Due to the large decrease in performance, fair locks should be used only in necessary circumstances.

Condition Variables

- If a thread that holds the lock on an object determines that it cannot continue with its task until some condition is satisfied, the thread can wait on a condition variable.
- This removes the thread from contention for the processor by placing it in a wait queue for the condition variable and releases the lock on the object.
- Condition variables must be associated with a `Lock` and are created by invoking `Lock` method `newCondition`, which returns an object that implements the `Condition` interface.
- To wait on a condition variable, the thread can call the `Condition`'s `await()` method (see Life Cycle of a thread in previous set of notes).

Condition Variables (cont.)

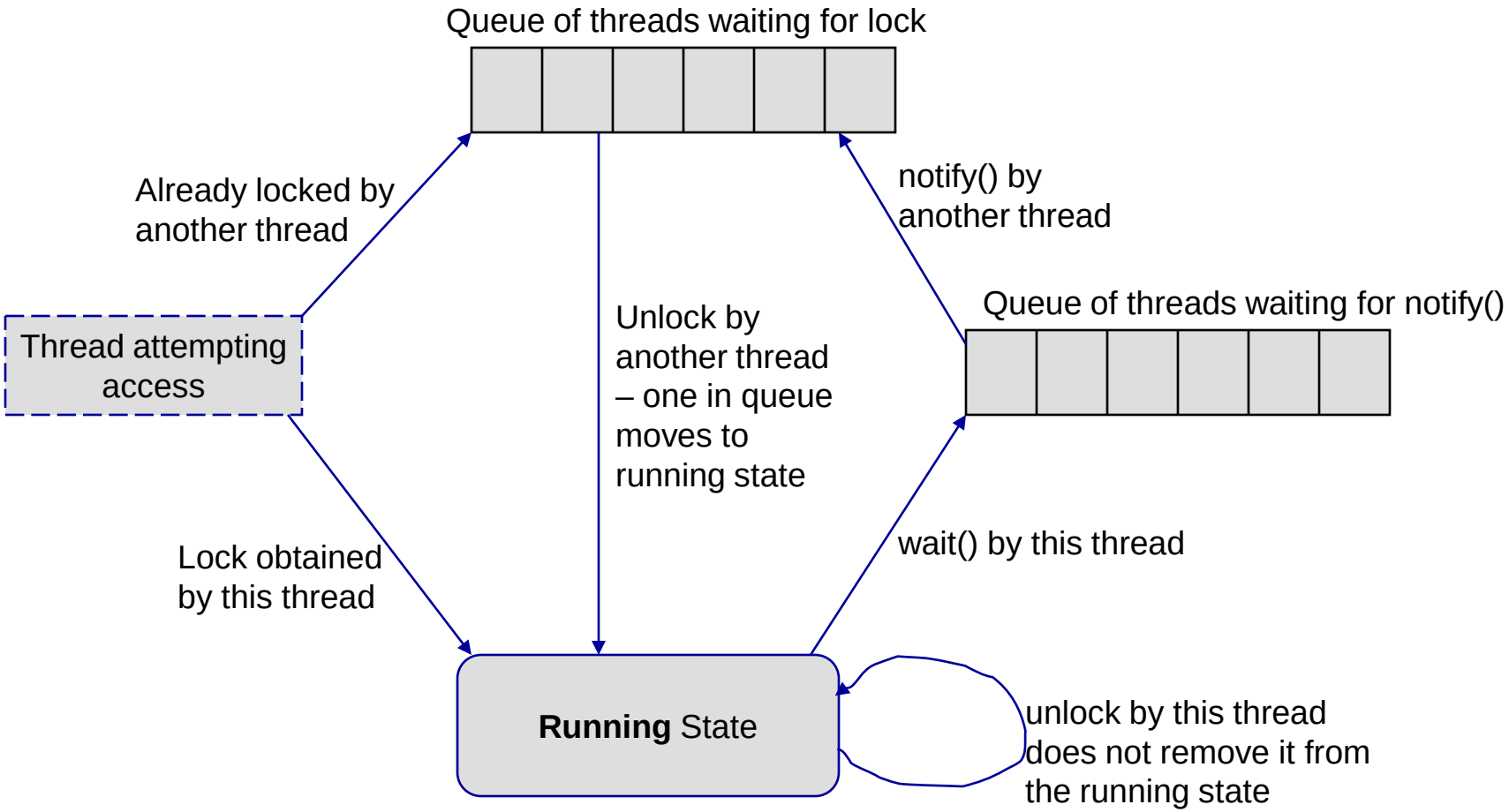
- Invoking the `await()` method, immediately releases the associated `Lock` and places the thread in the wait state for that `Condition`. Other threads can then try to obtain the `Lock`.

- When a runnable thread completes a task and determines that the waiting thread can now continue, the runnable thread can call Condition method `signal()` to allow a thread in that Condition's wait queue to return to the runnable state. At this point, the thread that transitioned from the wait state to the runnable state can attempt to reacquire the Lock on the object. Of course there is no guarantee that it will be able to complete its task this time and the cycle may repeat.

Condition Variables (cont.)

- If multiple threads are in a `Condition`'s wait queue when a `signal()` is invoked, the default implementation of `Condition` signals the longest-waiting thread to move to the runnable state.
- If a thread calls `Condition` method `signalAll()`, then all of the threads waiting for that condition move to the runnable state and become eligible to reacquire the `Lock`.
- When a thread is finished with a shared object, it must invoke method `unlock()` to release the `Lock`.

Thread States With Synchronization



Deadlock

- Deadlock will occur when a waiting thread (call it thread 1) cannot proceed because it is waiting (either directly or indirectly) for another thread (call it thread 2) to proceed., while simultaneously thread 2 cannot proceed because it is waiting (either directly or indirectly) for thread 1 to proceed.
- When multiple threads manipulate a shared object using locks, ensure that if one thread invokes `await` to enter the wait state for a condition variable, a separate thread eventually will invoke method `signal` to transition the waiting thread on the condition variable back to the runnable state.
  - If multiple threads may be waiting on the condition variable, a separate thread can invoke method `signalAll` as a safeguard to ensure that all of the waiting threads have another opportunity to perform their tasks.

Producer/Consumer Problem

Threads 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 an application reads data from the shared object.
  - Common examples are print spooling, copying data onto CDs, etc.
- In a multithreaded producer/consumer relationship, a producer thread generates data and places it in a shared object called a buffer. A consumer thread reads data from the buffer.
- What we want to consider first is how logic errors can arise if we do not synchronize access among multiple threads manipulating shared data.

Producer/Consumer w/o Synchronization

- The following example sets up a producer and consumer thread utilizing a shared buffer (code is on the webpage). The producer thread generates the integer numbers from 1 to 10, placing the values in the shared buffer. The consumer process reads the values in the buffer and prints the sum of all values consumed.
- Each value the producer thread writes into the buffer should be consumed exactly once by the consumer thread. However, the threads in this example are not synchronized.
  - This means that data can be lost if the producer writes new data into the buffer before the consumer has consumed the previous value.
  - Similarly, data can be incorrectly duplicated if the consumer thread consumes data again before the producer thread has produced the next value.

Producer/Consumer w/o Synchronization
(cont.)

- Since the producer thread will produce the values from 1 to 10, the correct sum that should be 55.
- The consumer process will arrive at this value only if each item produced by the producer thread is consumed exactly once by the consumer thread. No values are missed and none are consumed twice.
- I've set it up so that each thread writes to the screen what is being produced and what is being consumed.
- Note: the producer/consumer threads are put to sleep for a random interval between 0 and 3 seconds to emphasize the fact that in multithreaded applications, it is unpredictable when each thread will perform its task and for how long it will perform the task when it has a processor.

```
// Producer's run method stores the values 1 to 10 in buffer.
import java.util.Random;
public class Producer implements Runnable{
   private static Random generator = new Random();
   private Buffer sharedLocation; // reference to shared object

   // constructor
   public Producer( Buffer shared )  {
      sharedLocation = shared;
   } // end Producer constructor

   // store values from 1 to 10 in sharedLocation
   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 ) ); // sleep thread
            sharedLocation.set( count ); // set value in buffer
            sum += count; // increment sum of values
            System.out.printf( "\t%2d\n", sum );
         } // end try
         // if sleeping thread interrupted, print stack trace
         catch ( InterruptedException exception ) {
            exception.printStackTrace();
         } // end catch
      } // end for

      System.out.printf( "\n%s\n%s\n", "Producer done producing.",
         "Terminating Producer." );
   } // end method run
} // end class Producer
```

Producer Thread Class

Randomly sleep the thread for up to 3 seconds

```
// Consumer's run method loops ten times reading a value from buffer.
import java.util.Random;
public class Consumer implements Runnable {
   private static Random generator = new Random();
   private Buffer sharedLocation; // reference to shared object
   // constructor
   public Consumer( Buffer shared ) {
      sharedLocation = shared;
   } // end Consumer constructor
   // read sharedLocation's value four times and sum the values
   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.get();
            System.out.printf( "\t\t\t%2d\n", sum );
         } // end try
         // if sleeping thread interrupted, print stack trace
         catch ( InterruptedException exception ) {
            exception.printStackTrace();
         } // end catch
      } // end for
      System.out.printf( "\n%s %d.\n%s\n",
         "Consumer read values totaling", sum, "Terminating Consumer." );
   } // end method run
} // end class Consumer
```

Consumer Thread Class

Randomly sleep the thread for up to 3 seconds

```
// Buffer interface specifies methods called by Producer and Consumer.
public interface Buffer {
   public void set( int value ); // place int value into Buffer (WRITE)
   public int get(); // return int value from Buffer (READ)
} // end interface Buffer
```

Buffer Interface

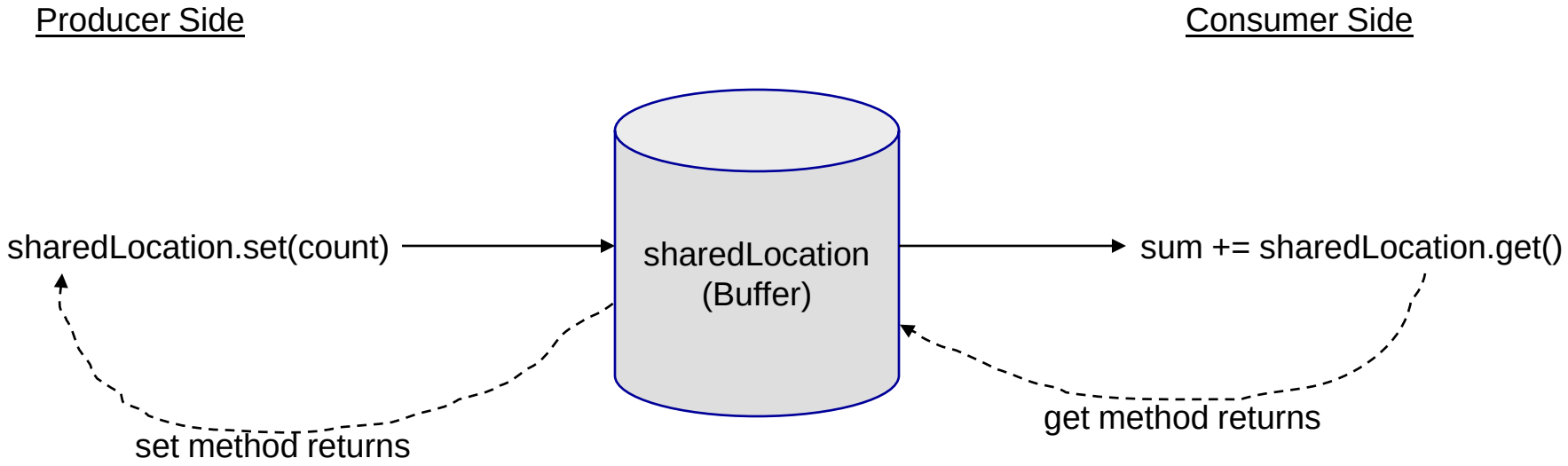
```
// UnsynchronizedBuffer represents a single shared integer.
public class UnsynchronizedBuffer implements Buffer {
   private int buffer = -1; // shared by producer and consumer threads
   // place value into buffer
   public void set( int value )   {
      System.out.printf( "Producer writes\t%2d", value );
      buffer = value;
   } // end method set

   // return value from buffer
   public int get()   {
      System.out.printf( "Consumer reads\t%2d", buffer );
      return buffer;
   } // end method get
} // end class UnsynchronizedBuffer
```

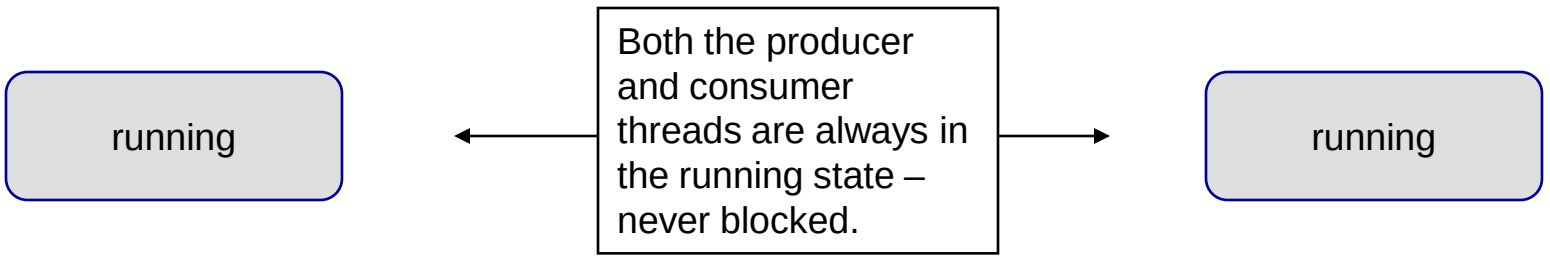
Unsynchronized Buffer Class

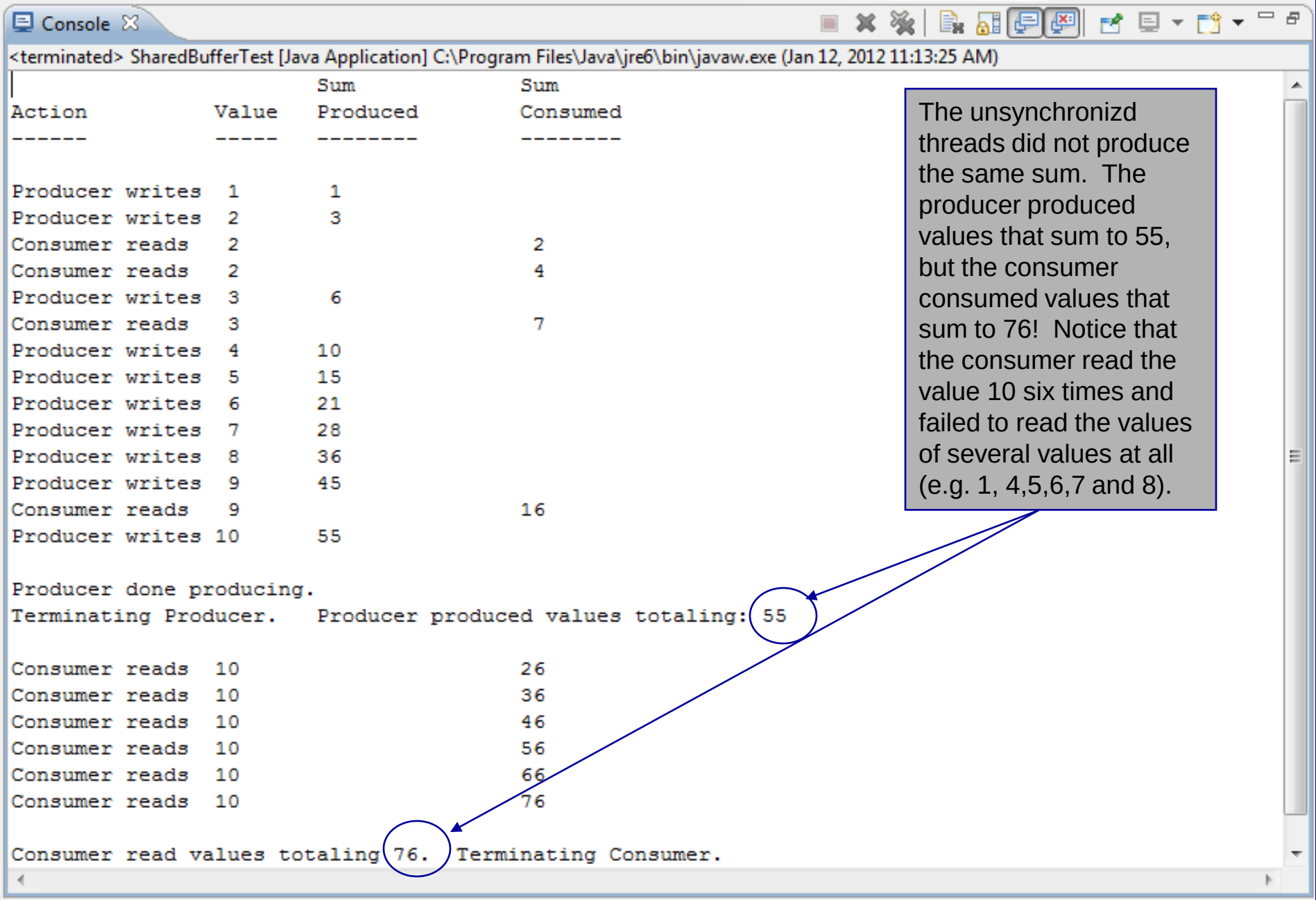
Producer/Consumer Driver Class

```
// Application shows two threads manipulating an unsynchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SharedBufferTest {
   public static void main( String[] args ){
      // create new thread pool with two threads
      ExecutorService application = Executors.newFixedThreadPool( 2 );

      // create UnsynchronizedBuffer to store ints
      Buffer sharedLocation = new UnsynchronizedBuffer();
      System.out.println( "     \t\t     \tSum     \tSum");
      System.out.println( "Action\t\tValue\tProduced\tConsumed" );
      System.out.println( "------\t\t-----\t--------\t--------\n" );

      // try to start producer and consumer giving each of them access to SharedLocation
       try  {
         application.execute( new Producer( sharedLocation ) );
         application.execute( new Consumer( sharedLocation ) );
      } // end try
      catch ( Exception exception ) {
         exception.printStackTrace();
      } // end catch

      application.shutdown(); // terminate application when threads end
   } // end main
} // end class SharedBufferTest
```


Unsynchronized Case






Console
<terminated> SharedBufferTest [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 12, 2012 11:13:25 AM)
Action Value Sum Produced Sum Consumed
Producer writes 1 1
Producer writes 2 3
Consumer reads 2 2
Consumer reads 2 4
Producer writes 3 6
Consumer reads 3 7
Producer writes 4 10
Producer writes 5 15
Producer writes 6 21
Producer writes 7 28
Producer writes 8 36
Producer writes 9 45
Consumer reads 9 16
Producer writes 10 55
Producer done producing.
Terminating Producer. Producer produced values totaling: 55
Consumer reads 10 26
Consumer reads 10 36
Consumer reads 10 46
Consumer reads 10 56
Consumer reads 10 66
Consumer reads 10 76
Consumer read values totaling 76. Terminating Consumer.
The unsynchronizd threads did not produce the same sum. The producer produced values that sum to 55, but the consumer consumed values that sum to 76! Notice that the consumer read the value 10 six times and failed to read the values of several values at all (e.g. 1, 4,5,6,7 and 8).

Console

<terminated> SharedBufferTest [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 12, 2012 11:16:21 AM)

```
Action          Value    Produced         Consumed
------          -----    --------         --------

Producer writes  1         1
Consumer reads   1                          1
Consumer reads   1                          2
Producer writes  2         3
Consumer reads   1                          4
Producer writes  3         6
Consumer reads   3                          7
Producer writes  4        10
Producer writes  5        15
Consumer reads   5                         12
Consumer reads   5                         17
Producer writes  6        21
Producer writes  7        28
Consumer reads   7                         24
Producer writes  8        36
Consumer reads   8                         32
Producer writes  9        45
Producer writes 10        55

Producer done producing.
Terminating Producer.   Producer produced values totaling: 55

Consumer reads  10                         42
Consumer reads  10                         52

Consumer read values totaling 52.  Terminating Consumer.
```

In this execution, the sum produced by the consumer is closer but still inaccurate because the consumer read the values of 1, 5 and 10 two times and failed to read the values of 2, 4, 6, and 9 at all.

```
// SynchronizedBuffer synchronizes access to a single shared integer.
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;
public class SynchronizedBuffer implements Buffer
{
  // Lock to control synchronization with this buffer
  private Lock accessLock = new ReentrantLock();
  // condition variables to control reading and writing
  private Condition canWrite = accessLock.newCondition();
  private Condition canRead = accessLock.newCondition();

  private int buffer = -1; // shared by producer and consumer threads
  private boolean occupied = false; // whether buffer is occupied
  // place int value into buffer
  public void set( int value )
  {
    accessLock.lock(); // lock this object
    // output thread information and buffer information, then wait
    try
    {
      // while buffer is not empty, place thread in waiting state
      while ( occupied )
      {
        System.out.println( "Producer tries to write." );
        displayState( "Buffer full. Producer waits." );
        canWrite.await(); // wait until buffer is empty
      } // end while
```

Synchronized Buffer Class

No fairness policy needed since only a single producer thread and single consumer thread

Condition variables on the lock. Condition canWrite contains a queue for threads waiting to write while the buffer is full. If the buffer is full the Producer calls method await on this condition. When the Consumer reads data from a full buffer, it calls method signal on this Condition. Condition canRead contains a queue for threads waiting while the buffer is empty. If the buffer is empty the Consumer calls method await on this Condition. When the Producer writes to the empty buffer, it will call method signal on this Condition.

Acquire lock

```
buffer = value; // set new buffer value
      // indicate producer cannot store another value
      // until consumer retrieves current buffer value
      occupied = true;
      displayState( "Producer writes " + buffer );
      // signal thread waiting to read from buffer
      canRead.signal();
   } // end try
   catch ( InterruptedException exception )    {
      exception.printStackTrace();
   } // end catch
   finally   {
      accessLock.unlock(); // unlock this object
   } // end finally
} // end method set

// return value from buffer
public int get()   {
   int readValue = 0; // initialize value read from buffer
   accessLock.lock(); // lock this object
   // output thread information and buffer information, then wait
   try      {
      // while no data to read, place thread in waiting state
      while ( !occupied )   {
         System.out.println( "Consumer tries to read." );
         displayState( "Buffer empty. Consumer waits." );
        canRead.await(); // wait until buffer is full
      } // end while
```

Signal Consumer thread that a value has been produced and can be read.

Unlock object before exiting method

Acquire lock on the buffer

Consumer must wait until a value has been produced by the Producer. Await signal by Producer

```
      // indicate that producer can store another value
         // because consumer just retrieved buffer value
         occupied = false;
         readValue = buffer; // retrieve value from buffer
         displayState( "Consumer reads " + readValue );
         // signal thread waiting for buffer to be empty
         canWrite.signal();
      } // end try
      // if waiting thread interrupted, print stack trace
      catch ( InterruptedException exception )      {
         exception.printStackTrace();
      } // end catch
      finally    {
         accessLock.unlock(); // unlock this object
      } // end finally

      return readValue;
   } // end method get

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

Signal waiting Producer that the buffer is empty and it can write

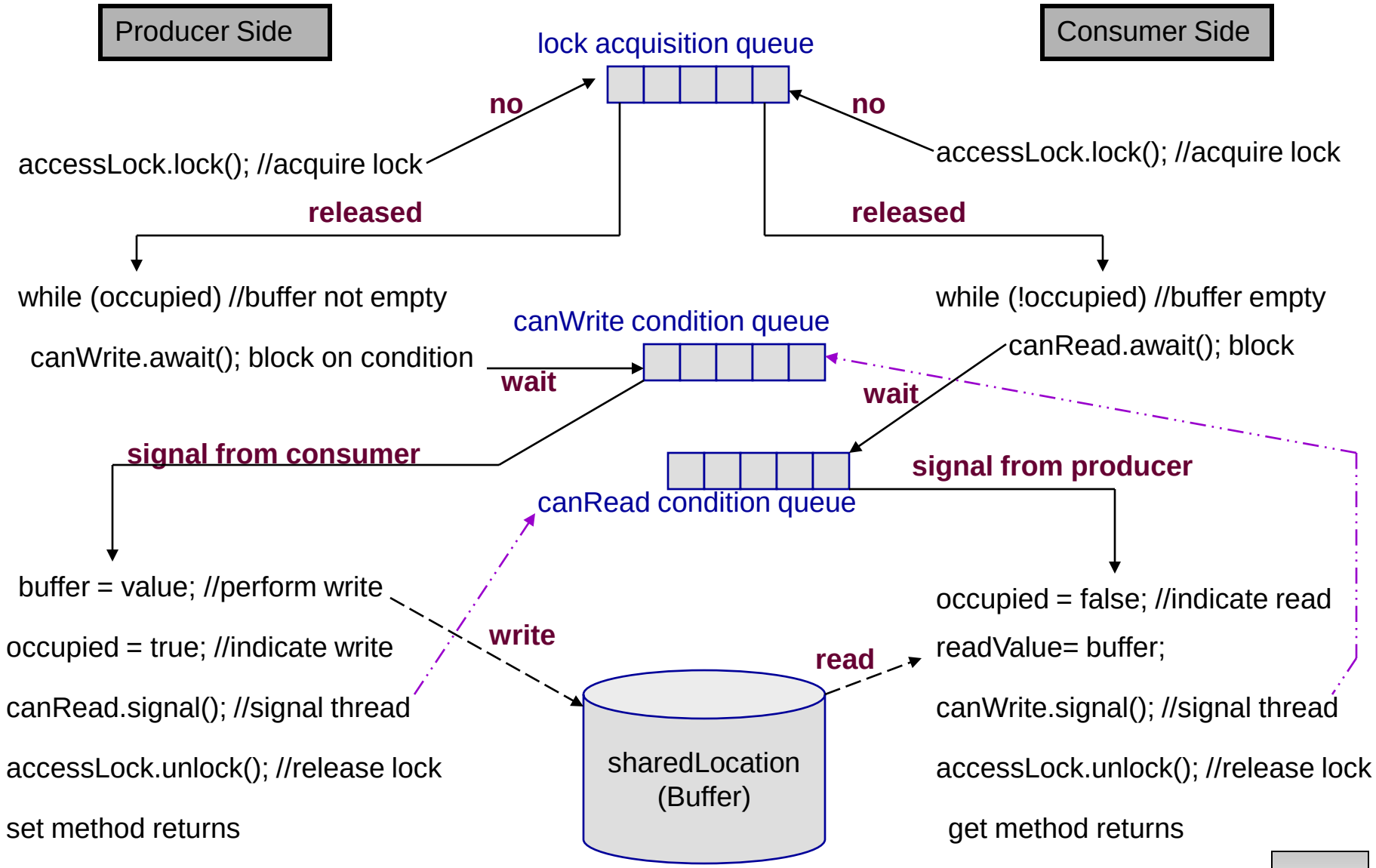
Make sure lock is released

Driver Class For Illustrating Synchronization In Producer/Consumer Problem

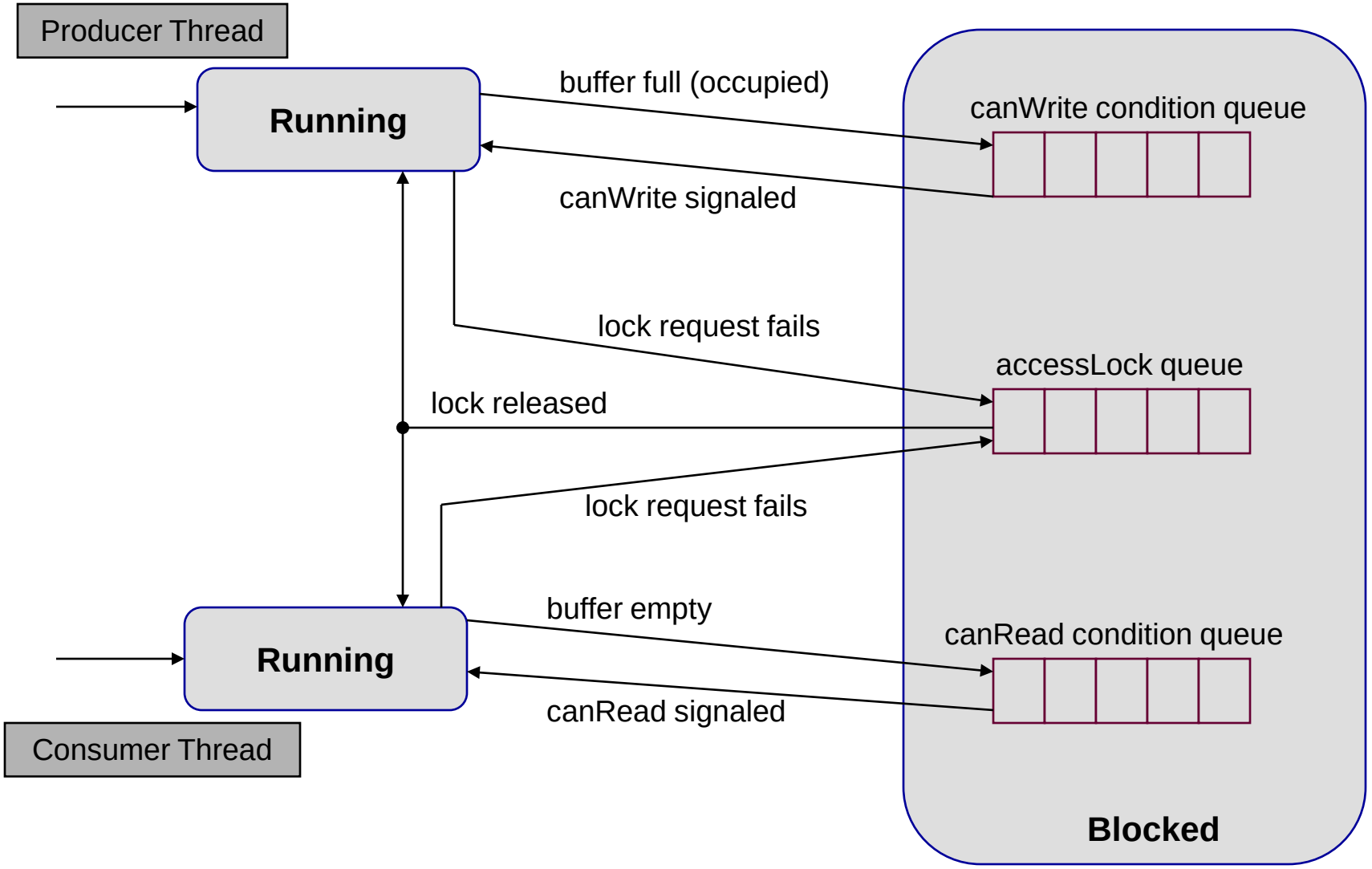
```
// Application shows two threads manipulating a synchronized buffer.
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class SharedBufferTest2
{
  public static void main( String[] args )
  {
    // create new thread pool with two threads
    ExecutorService application = Executors.newFixedThreadPool( 2 );

    // create SynchronizedBuffer to store ints
    Buffer sharedLocation = new SynchronizedBuffer();
    System.out.println("Using Standard Locking");
    System.out.printf( "%-40s%s\t\t%s\n%-40s%s\n\n", "Operation",
      "Buffer Contents", "Occupied", "---------", "---------------\t\t--------" );

    try   {  // try to start producer and consumer
      application.execute( new Producer( sharedLocation ) );
      application.execute( new Consumer( sharedLocation ) );
    } // end try
    catch ( Exception exception )
    {
      exception.printStackTrace();
    } // end catch

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

Only change between SharedBufferTest for unsynchronized version

Synchronized Case



State Diagram – Synchronized Version



Console

<terminated> SharedBufferTest2 [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 12, 2012 11:25:24 AM)

```
Using Standard Locking
Operation                          Buffer Contents          Occupied
---------                          ---------------          --------

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
Consumer tries to read.
Buffer empty. Consumer waits.      2                          false
Producer writes 3                  3                          true
Consumer reads 3                   3                          false
Producer writes 4                  4                          true
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
Producer tries to write.
Buffer full. Producer waits.       8                          true
Consumer reads 8                   8                          false
```


```
Producer tries to write.
Buffer full. Producer waits.        8                   true
Consumer reads 8                    8                   false
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.
Producer produced values totaling: 55
```

Both the Producer and Consumer threads produced the same sum – synchronized threads

Monitors and Monitor Locks

- Another way to perform synchronization is to use Java's built-in monitors. Every object has a monitor. Strictly speaking, the monitor is not allocated unless it is used.
- A monitor allows one thread at a time to execute inside a synchronized statement on the object. This is accomplished by acquiring a lock on the object when the program enters the synchronized statement.

```
synchronized (object)
{
        statements
} //end synchronized statement
```

  - Where *object* is the object whose monitor lock will be acquired.

- If there are several synchronized statements attempting to execute on an object at the same time, only one of them may be active on the object at once – all the other threads attempting to enter a synchronized statement on the same object are placed into the blocked state.

Mutual Exclusion Over a Block of Statements

- When a synchronized statement finishes executing, the monitor lock on the object is released and the highest priority blocked thread attempting to enter a synchronized statement proceeds.
- Applying mutual exclusion to a block of statements rather than to an entire class or an entire method is handled in much the same manner, by attaching the keyword synchronized before a block of code.
- You must explicitly mention in parentheses the object whose lock must be acquired before the block can be entered.
- The next page illustrates the steam boiler pressure gauge problem using a synchronized statement block to control the threads access to the pressure gauge.

syncPressure.java | syncSteamBoiler.java

```
//thread class to raise the pressure in the Boiler
class syncPressure extends Thread {
   static Object O = new Object();
   void RaisePressure() {
      synchronized(O) {
      if (syncSteamBoiler.pressureGauge < syncSteamBoiler.safetyLimit-15) {
         //wait briefly to simulate some calculations
         try {sleep(200); } catch (Exception e) { }
         syncSteamBoiler.pressureGauge+= 15; //raise the pressure 15 psi
         System.out.println("Thread " + this.getName() + " finds pressure within limits - increases
       }
       else
         System.out.println("Thread" + this.getName() + " finds pressure too high - do nothing");
      }//end synchronized object
   }
   public void run() {
      RaisePressure();  //this thread is to raise the pressure
   }
}
```

Synchronized statement requires an Object to lock.

Synchronized block

syncPressure.java | syncSteamBoiler.java

```
//thread class to raise the pressure in the Boiler
class syncPressure extends Thread {
   static Object O = new Object();
   void RaisePressure() {
       synchronized(O) {
       if (syncSteamBoiler.pressureGauge < syncSteamBoiler.safetyLimit-15) {
          //wait briefly to simulate some calculations
          try {sleep(200); } catch (Exception e) { }
          syncSteamBoiler.pressureGauge+= 15; //raise the pressure 15 psi
          System.out.println("Thread " + this.getName() + " finds pressure within limits - increases
        }
        else
          System.out.println("Thread" + this.getName() + " finds pressure too high - do nothing");
       }//end synchronized object
```

Console

Show Console When Standard

```
<terminated> syncSteamBoiler [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 12, 2012 12:12:30 PM)
Thread Thread-0 finds pressure within limits - increases pressure
Thread Thread-9 finds pressure within limits - increases pressure
Thread Thread-8 finds pressure within limits - increases pressure
ThreadThread-7 finds pressure too high - do nothing
ThreadThread-6 finds pressure too high - do nothing
ThreadThread-5 finds pressure too high - do nothing
ThreadThread-4 finds pressure too high - do nothing
ThreadThread-3 finds pressure too high - do nothing
ThreadThread-2 finds pressure too high - do nothing
ThreadThread-1 finds pressure too high - do nothing

Gauge reads 45, the safe limit is 50...System OK!
```


Monitors and Monitor Locks (cont.)

- Java also allows synchronized methods. A synchronized method is equivalent to a synchronized statement enclosing the entire body of a method.
- If a thread obtains the monitor lock on an object and then discovers that it cannot continue with its task until some condition is satisfied, the thread can invoke Object method `wait`, releasing the monitor lock on the object. This will place the thread in the wait state.
- When a thread executing a synchronized statement completes or satisfies the condition on which another thread may be waiting, it can invoke Object method `notify` to allow a waiting thread to transition to the blocked state again.

Caution When Using Synchronization

- As with any multi-threaded application, care must be taken when using synchronization to achieve the desired effect and not introduce some serious defect in the application.
- Consider the variation of the pressure gauge example that we've been dealing with on the following page. Study the code carefully and try to determine if it will achieve the same effect as the previous version of the code.
- Is it correct? Why or why not?

syncPressure.java | syncSteamBoil

Does this code correctly synchronize the pressure gauge reading threads?

```
//thread class to raise the pressure in the Boiler
class syncPressure extends Thread {
   //static Object O = new Object();
   synchronized void RaisePressure() {
       if (syncSteamBoiler.pressureGauge < syncSteamBoiler.safetyLimit-15) {
          //wait briefly to simulate some calculations
          try {sleep(100); } catch (Exception e) { }
          syncSteamBoiler.pressureGauge+= 15; //raise the pressure 15 psi
          System.out.println("Thread " + this.getName() + " finds pressure within limits - increases
       }
       else
          System.out.println("Thread" + this.getName() + " finds pressure too high - do nothing");
   }
   public void run() {
      RaisePressure();  //this thread is to raise the pressure
   }
}
```

No! The "this" object is one of the 10 different threads that are created. Each thread will successfully grab its own lock, and there will be no exclusion between the different threads.

Synchronization excludes threads working on the *same object*; it does not synchronize the *same method* on different objects!

syncPressure.java | syncSteamBoiler.java

```
//thread class to raise the pressure in the Boiler
class syncPressure extends Thread {
   //static Object O = new Object();
   synchronized void RaisePressure() {
       if (syncSteamBoiler.pressureGauge < syncSteamBoiler.safetyLimit-15) {
          //wait briefly to simulate some calculations
          try {sleep(100); } catch (Exception e) { }
          syncSteamBoiler.pressureGauge+= 15; //raise the pressure 15 psi
          System.out.println("Thread " + this.getName() + " finds pressure within limits - increases
        }
        else
          System.out.println("Thread" + this.getName() + " finds pressure too high - do nothing");
   }
   public void run() {
```

Console

<terminated> syncSteamBoiler [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jan 12, 2012 11:43:47 AM)

```
Thread Thread-1 finds pressure within limits - increases pressure
Thread Thread-4 finds pressure within limits - increases pressure
Thread Thread-3 finds pressure within limits - increases pressure
Thread Thread-0 finds pressure within limits - increases pressure
Thread Thread-9 finds pressure within limits - increases pressure
Thread Thread-5 finds pressure within limits - increases pressure
Thread Thread-2 finds pressure within limits - increases pressure
Thread Thread-6 finds pressure within limits - increases pressure
Thread Thread-8 finds pressure within limits - increases pressure
Thread Thread-7 finds pressure within limits - increases pressure

Gauge reads 135, the safe limit is 50...B O O M ! ! !
```

NO!