LOGBOOK

Algorithms – Processes and data

J.Pritchard U1661665
U1661665@unimail.hud.ac.uk

Practical 9 (Week 13)

(Model) Question 1: Implement the run() method so that when a counter thread is run it will start a while loop to run through all values of the counter.

Code Listing.

Counter.

```
package counter;

public class Counter extends Thread
{
    ...
    ...

    /**
     * Start the counter by setting it to the initial value
     **/
    public void startCount() {
       delay();
        counter = from;
        if (tracingOn) System.out.println(getName() + " has started: " + counter);
    }

    /**
     * Increment the counter.
     **/
    public void stepCount() {
        delay();
        counter += step;
        if (tracingOn) System.out.println(getName() + " has stepped: " + counter);
    }

    /**
     * Check whether the counter has passed its limiting value.  If the increment is positive
     * the counter has passed its limit if it is greater than the limit.  If the increment is
     * less than zero the counter must be lower than its limit.
     * @return true iff this counter has passed its limiting value.
     **/
    public boolean isFinished() {
       delay();
```

```
        boolean finished =
          (step > 0 && counter >= limit) || (step < 0 && counter <= limit);
        if (tracingOn && finished) System.out.println(getName() + " has finished: " + counter);
        return finished;
    }

    /**
     * Run this counter.
     */
    public void run() {
        //Initialize the counter.
        this.startCount();

        //While this counter is not finished.
        while(!this.isFinished())
        {
            //Step the counter.
            this.stepCount();
        }

    }
}
```

ThreadHashSet

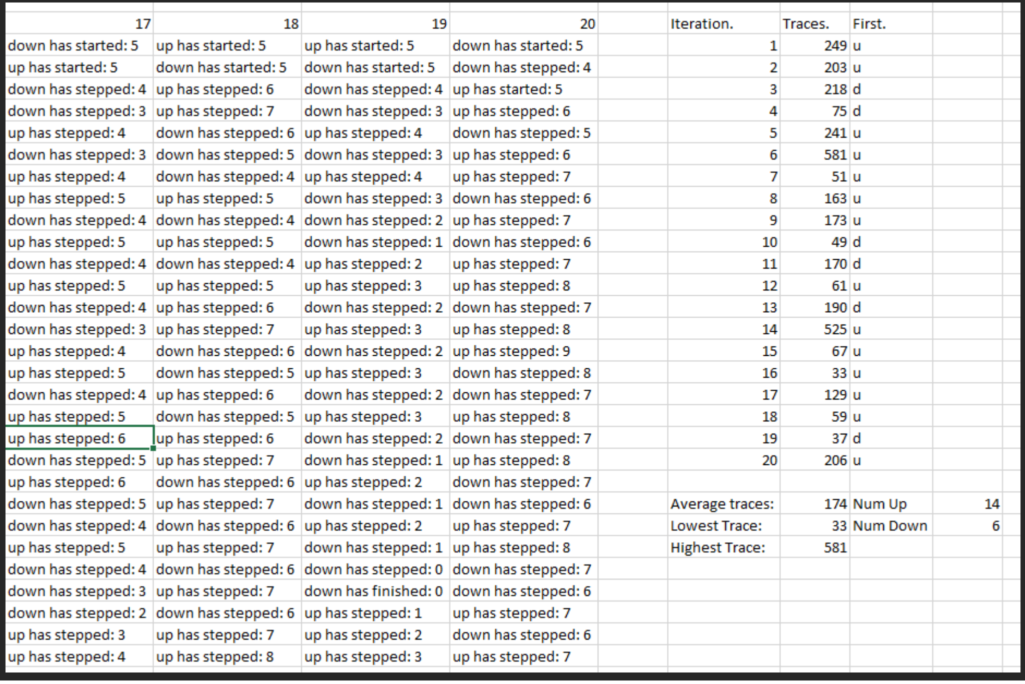
```java
package counter;

import java.util.HashSet;

/**
 * An implementation of thread sets.
 * A thread set is a set of Threads which provides a facility for running all the threads in the set concurrently.
 *
 * @author Hugh Osborne
 * @version January 2019
 */

public class ThreadHashSet<T extends Thread> extends HashSet<T> implements ThreadSet<T>{

   /**
    * Run all the threads in this thread set in parallel, and wait for them to finish.
    * @throws InterruptedException if an interrupt occurs while witing for the trheads to finish.
    */
   @Override
   public void runSet() throws InterruptedException {
      // Implement runSet here.
      // runSet() should start up all the threads in this set, and then wait for them to finish.

        //Start each thread stored in the set.
      for(Thread thread : this)
        {
            thread.start();
        }

        //Wait for each thread in the set.
        for(Thread thread : this)
        {
            thread.join();
        }
   }

}
```

(Demo code) Question 2: The main class contains a method demonstrating the use of counters and threadsets.

Try running the main method a few times and observe the counter’s behaviour. Try editing the main method to change the Counter delay to low and high delays, and try running the tests again. You may observe a difference in behaviour. If you do, what is this difference, and why do you think it occurs?

Testing

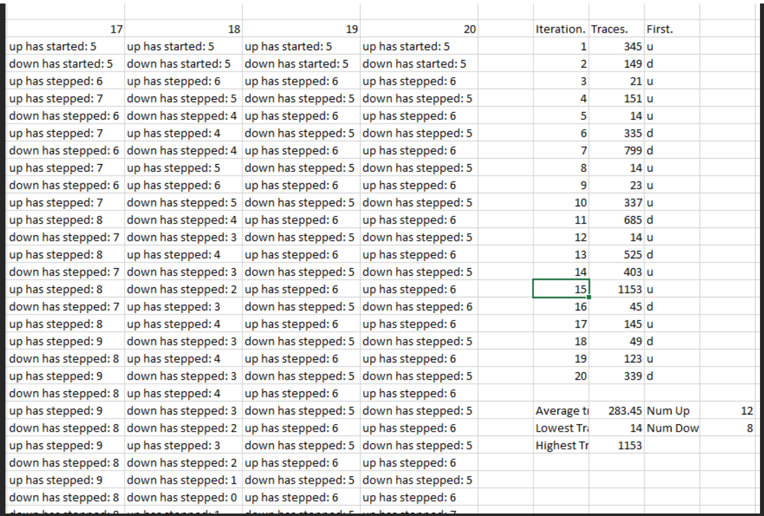
```
* Important here: delay is not 0.1 for everything, it is between 0.0 and 0.1, chosen randomly.
*  . The behaviour observed is that eventually one counter overpowers the other in a concurrency 'tug of war'.
*  . Obviously this means that the other counter is then free to count to its goal value.
*  . Both counters are put in equal starting positions with the same difference between their starting values and their goal values.
*  . The above means that it should be an even distribution of which counter reaches its goal value first.
*  . This is because the random time generation has no bias towards either counter, which results in a 0.5/0.5 chance for each counter to perform its
operation.
*  .
```



17	18	19	20		Iteration.	Traces.	First.	
down has started: 5	up has started: 5	up has started: 5	down has started: 5		1	249	u	
up has started: 5	down has started: 5	down has started: 5	down has stepped: 4		2	203	u	
down has stepped: 4	up has stepped: 6	down has stepped: 4	up has started: 5		3	218	d	
down has stepped: 3	up has stepped: 7	down has stepped: 3	up has stepped: 6		4	75	d	
up has stepped: 4	down has stepped: 6	up has stepped: 4	down has stepped: 5		5	241	u	
down has stepped: 3	down has stepped: 5	down has stepped: 3	up has stepped: 6		6	581	u	
up has stepped: 4	down has stepped: 4	up has stepped: 4	up has stepped: 7		7	51	u	
up has stepped: 5	up has stepped: 5	down has stepped: 3	down has stepped: 6		8	163	u	
down has stepped: 4	down has stepped: 4	down has stepped: 2	up has stepped: 7		9	173	u	
up has stepped: 5	up has stepped: 5	down has stepped: 1	down has stepped: 6		10	49	d	
down has stepped: 4	down has stepped: 4	up has stepped: 2	up has stepped: 7		11	170	d	
up has stepped: 5	up has stepped: 5	up has stepped: 3	up has stepped: 8		12	61	u	
down has stepped: 4	up has stepped: 6	down has stepped: 2	down has stepped: 7		13	190	d	
down has stepped: 3	up has stepped: 7	up has stepped: 3	up has stepped: 8		14	525	u	
up has stepped: 4	down has stepped: 6	down has stepped: 2	up has stepped: 9		15	67	u	
up has stepped: 5	down has stepped: 5	up has stepped: 3	down has stepped: 8		16	33	u	
down has stepped: 4	up has stepped: 6	down has stepped: 2	down has stepped: 7		17	129	u	
up has stepped: 5	down has stepped: 5	up has stepped: 3	up has stepped: 8		18	59	u	
up has stepped: 6	up has stepped: 6	down has stepped: 2	down has stepped: 7		19	37	d	
down has stepped: 5	up has stepped: 7	down has stepped: 1	up has stepped: 8		20	206	u	
up has stepped: 6	down has stepped: 6	up has stepped: 2	down has stepped: 7					
down has stepped: 5	up has stepped: 7	down has stepped: 1	down has stepped: 6		Average traces:	174	Num Up	14
down has stepped: 4	down has stepped: 6	up has stepped: 2	up has stepped: 7		Lowest Trace:	33	Num Down	6
up has stepped: 5	up has stepped: 7	down has stepped: 1	up has stepped: 8		Highest Trace:	581		
down has stepped: 4	down has stepped: 6	down has stepped: 0	down has stepped: 7					
down has stepped: 3	up has stepped: 7	down has finished: 0	down has stepped: 6					
down has stepped: 2	down has stepped: 6	up has stepped: 1	up has stepped: 7					
up has stepped: 3	up has stepped: 7	up has stepped: 2	down has stepped: 6					
up has stepped: 4	up has stepped: 8	up has stepped: 3	up has stepped: 7					

```
*  .
```

```
 *  .
 *  . The test data shows the results of 20 iterations of the standard main method.
 *  . There is a small possibility, given the equal starting positions, that the system could run in an endless loop of counters.
 *  . The above is demonstrated by the longest trace stack being 581, the closest the 20 iterations got to infinity.
 *  . The inverse, that a counter could perform all its steps before the other gets a chance to do anything is also true.
 *  . This can be mathematically proven:
 *  . The lowest millisecond bound is 0.0 * 1000 = 0
 *  . The highest millisecond bound is 0.1 * 1000 = 100
 *  . The random number generator simply chooses a number between these two millisecond values.
 *  . If counter A got given 100ms.
 *  . Then counter B could theoretically be given 5 steps of 1ms.
 *  . These 5ms < 100ms which means that counter B would complete all of its steps before A completes its first step.
 *  . This is comparable in probability to the infinity loop situation however, and is not achieved in the 20 iterations.
 *  . However, a close number was achieved by iteration 16, which only took 33 trace steps.
 *  .
 *  . One point of interest within this test data is in Iteration 10.
 *  . Within this iteration, the down counter reaches 0 however the up counter then steps before the down has a chance to finish its operation.
 *  . This demonstrates a consequence of concurrent programming with shared variables.
 *  . The demonstration being evident in the code that the up counter has stepped in the period between the step() and the start of the !isFinished() while
loop.
 *  . While the counter was stepped to 0, it was stepped back to 1 before the isFinished() method queried it.
 *  . The consequence demonstrated is the lack of predictability in the code.
 *  . In a larger system, this could cause a fatal error.
 *  . This exact consequence could be solved with the use of critical sections, but simply serves as a warning in this piece of code.
 *  .
 *  . Another piece of information that may be considered relevant is the ratio of iterations which finished their up counter first - 14:6.
 *  . At first glance this may suggest that the up counter has a bias on it to finish first.
 *  . This is false, and is simply a consequence of the low number of iterations performed on the test.
 *  . Given a higher number of iterations, the even distribution would be observed.
 *  .
```



17	18	19	20		Iteration.	Traces.	First.	
up has started: 5	up has started: 5	up has started: 5	up has started: 5		1	345	u	
down has started: 5	down has started: 5	down has started: 5	down has started: 5		2	149	d	
up has stepped: 6	up has stepped: 6	up has stepped: 6	up has stepped: 6		3	21	u	
up has stepped: 7	down has stepped: 5	down has stepped: 5	down has stepped: 5		4	151	u	
down has stepped: 6	down has stepped: 4	up has stepped: 6	up has stepped: 6		5	14	u	
up has stepped: 7	up has stepped: 4	down has stepped: 5	down has stepped: 5		6	335	d	
down has stepped: 6	down has stepped: 4	up has stepped: 6	up has stepped: 6		7	799	d	
up has stepped: 7	up has stepped: 5	down has stepped: 5	down has stepped: 5		8	14	u	
down has stepped: 6	up has stepped: 6	up has stepped: 6	up has stepped: 6		9	23	u	
up has stepped: 7	down has stepped: 5	down has stepped: 5	down has stepped: 5		10	337	u	
up has stepped: 8	down has stepped: 4	up has stepped: 6	up has stepped: 6		11	685	d	
down has stepped: 7	down has stepped: 3	down has stepped: 5	down has stepped: 5		12	14	u	
up has stepped: 8	up has stepped: 4	up has stepped: 6	up has stepped: 6		13	525	d	
down has stepped: 7	down has stepped: 3	down has stepped: 5	down has stepped: 5		14	403	u	
up has stepped: 8	down has stepped: 2	up has stepped: 6	up has stepped: 6		15	1153	u	
down has stepped: 7	up has stepped: 3	down has stepped: 5	down has stepped: 6		16	45	d	
up has stepped: 8	up has stepped: 4	up has stepped: 6	up has stepped: 6		17	145	u	
up has stepped: 9	down has stepped: 3	down has stepped: 5	down has stepped: 5		18	49	d	
down has stepped: 8	up has stepped: 4	up has stepped: 6	up has stepped: 6		19	123	u	
up has stepped: 9	down has stepped: 3	down has stepped: 5	down has stepped: 5		20	339	d	
down has stepped: 8	up has stepped: 4	up has stepped: 6	up has stepped: 6					
up has stepped: 9	down has stepped: 3	down has stepped: 5	down has stepped: 5		Average ti	283.45	Num Up	12
down has stepped: 8	down has stepped: 2	up has stepped: 6	up has stepped: 6		Lowest Tra	14	Num Dow	8
up has stepped: 9	up has stepped: 3	down has stepped: 5	down has stepped: 5		Highest Tr	1153		
down has stepped: 8	down has stepped: 2	up has stepped: 6	up has stepped: 6					
up has stepped: 9	down has stepped: 1	down has stepped: 5	down has stepped: 5					
down has stepped: 8	down has stepped: 0	up has stepped: 6	up has stepped: 6					

* .

* .

* . This test data does the same thing with the delay changed to 0.001

* . This means that the upper bound is now 0.001 * 1000 = 1ms.

* . the lower bound is unchanged, meaning the counter can either choose from 0 or 1 for the delay time.

* . This decreases the variance in the values for the delay and means that the probability a counter will be faster than the other is roughly 0.5

* .

* . This means that this test data is more accurate at showing the theories discussed with the previous test data.

* . The first one to look at will be the 'A versus B' theory where one counter steps all the way through before the other counter is able to step once.

* . This occurs three separate times in the 20 iterations, with all of them having the 'up' counter finish before 'down' has stepped once.

* . The issue might be raised, given this result, that it's impossible for counter A to step through all the way when the options are as small as 0 and 1ms.

* . This can be explained in terms of processor tick speed:

* . Counter A has had every step be given a delay time of 0ms, whereas the first step of B has been given a step time of 1ms.

* . Processes in each thread of a CPU are executed as fast as the tick speed of the hardware.

* . The delay value is here is irrelevant to that and simply emulates a latency based architecture.

* . So the conclusion drawn from this must be that the tick speed of the processor is high enough that a single thread can execute all 5 instructions in less than 1ms.

* . (This makes a lot of sense given the clock speed of modern CPUs being well in the GHz).

* .

* . The infinity theory is also very well demonstrated here - the largest number of iterations was a staggering 1153.

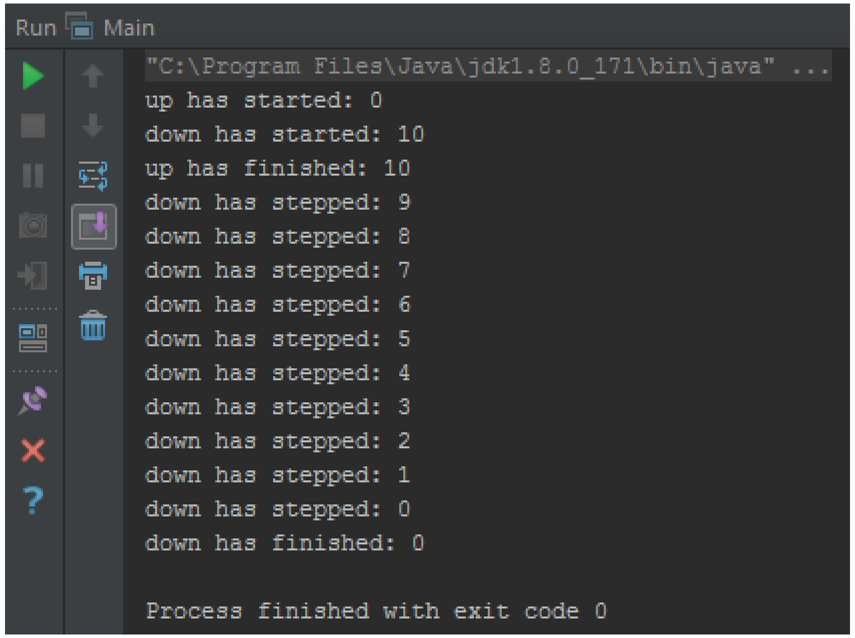
```
*  . The even distribution is somewhat better shown here as well, with 12:8 (up:down).
*  .
*  . I tried an iteration of testing where I allowed the maximum bound to be 1000ms.
*  . The first iteration of this took 30 minutes and showed exactly the same behaviour, so I won't be continuing this test run.
```

(Logbook) Question 3: Edit the main method so that the counters now try to count from 0 to 10, and from 10 to 0, in steps of +- 1.

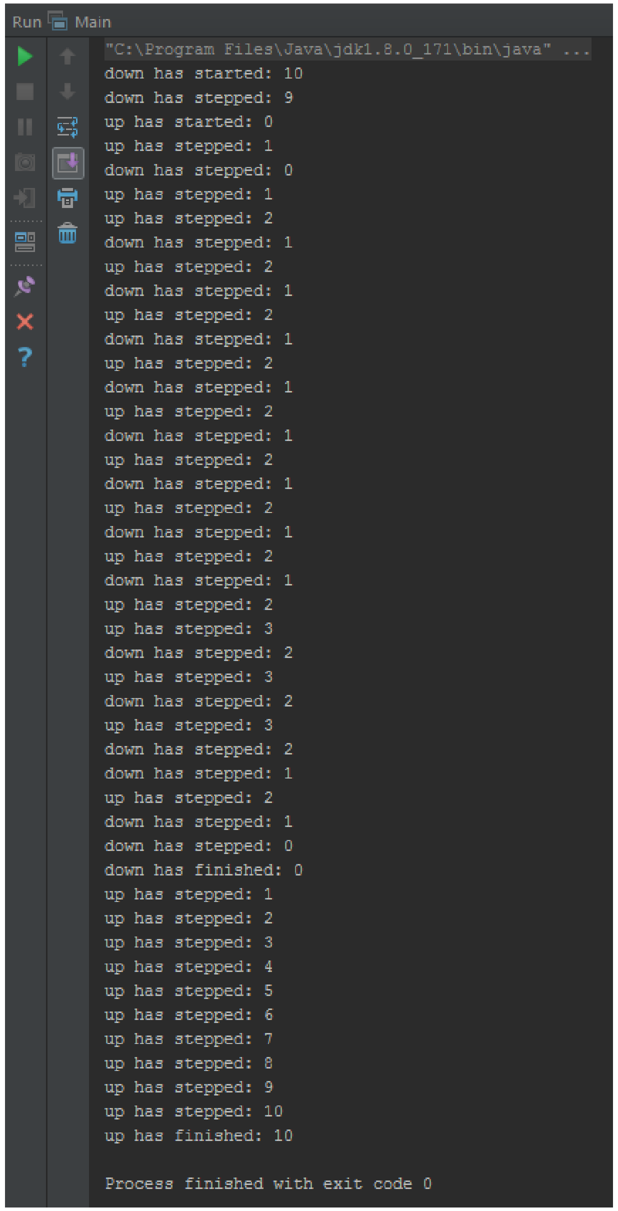
Code Listing.

```java
/**
 * A demonstration of the use and behaviour of Counters and ThreadSets.
 *
 * @author Hugh Osborne
 * @implNote Edited: Joshua Pritchard (U1661665)
 * @version January 2019
 */
public class Main {
    /**
     * Demonstrate the behaviour of counters and ThreadSets.  A thread set is populated with two counters, and
     * the thread set's runSet method is used to run the counters concurrently.
     *
     * @param args not used
     * @throws CounterException should not occur
     * @throws InterruptedException should not occur
     */
    public static void main(String[] args) throws CounterException, InterruptedException {
      /*
       * Create two counters (in a thread set), and then run them with tracing on, so that their
       * behaviour is visible.
       */
        ThreadSet<Counter> counters = new ThreadHashSet<>();  // will contain the counters
        counters.add(new Counter("up",0,10)); // counter "up" counts from 5 to 10
        counters.add(new Counter("down",10,0)); // counter "down" counts from  to 0
        Counter.traceOn(); // switch tracing on
        Counter.setDelay(0.1); // set a delay from 0.0 to 0.1 seconds
        counters.runSet(); // run the counters (concurrently)
    }
}
```

Some example results.



```
"C:\Program Files\Java\jdk1.8.0_171\bin\java" ...
up has started: 0
down has started: 10
up has finished: 10
down has stepped: 9
down has stepped: 8
down has stepped: 7
down has stepped: 6
down has stepped: 5
down has stepped: 4
down has stepped: 3
down has stepped: 2
down has stepped: 1
down has stepped: 0
down has finished: 0

Process finished with exit code 0
```



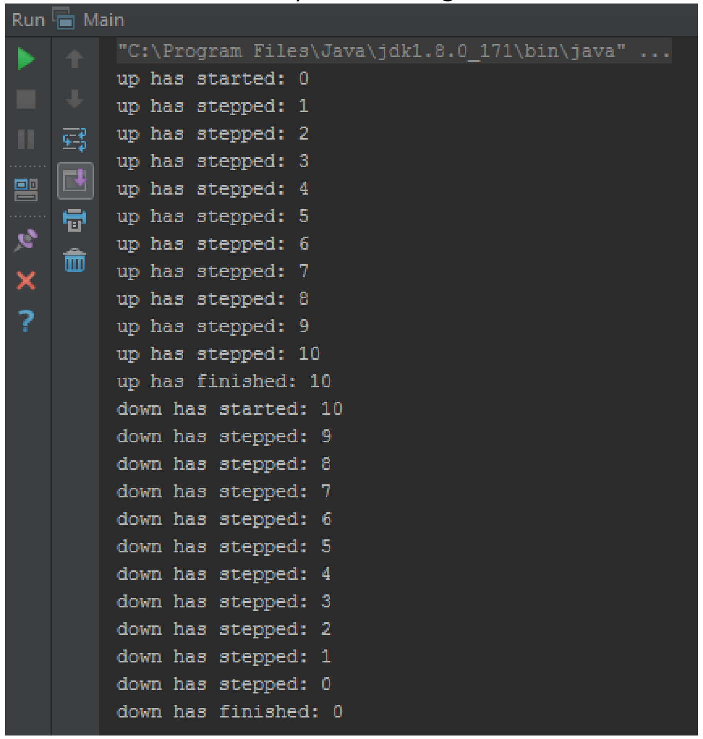
```
"C:\Program Files\Java\jdk1.8.0_171\bin\java" ...
down has started: 10
down has stepped: 9
up has started: 0
up has stepped: 1
down has stepped: 0
up has stepped: 1
up has stepped: 2
down has stepped: 1
up has stepped: 2
down has stepped: 1
up has stepped: 2
down has stepped: 1
up has stepped: 2
down has stepped: 1
up has stepped: 2
down has stepped: 1
up has stepped: 2
down has stepped: 1
up has stepped: 2
down has stepped: 1
up has stepped: 2
down has stepped: 1
up has stepped: 2
up has stepped: 3
down has stepped: 2
up has stepped: 3
down has stepped: 2
up has stepped: 3
down has stepped: 2
down has stepped: 1
up has stepped: 2
down has stepped: 1
down has stepped: 0
down has finished: 0
up has stepped: 1
up has stepped: 2
up has stepped: 3
up has stepped: 4
up has stepped: 5
up has stepped: 6
up has stepped: 7
up has stepped: 8
up has stepped: 9
up has stepped: 10
up has finished: 10

Process finished with exit code 0
```

Question 1: Will the test always terminate? I.e. is it certain that no matter how often you were to run the test it would always end in a finite length of time?

No.
The test is not guaranteed to end at all. There is a slim probability that the test will run for an infinite amount of time. This is because there are plenty of cases which can cause the test to get to a non-goal value and then have both counters acting in turn, one after the other. This results in an up-down-up-down…etc. behaviour that goes on for potentially infinite iterations. This is demonstrated and discussed in detail in the Demo Code above. The reason that this behaviour is not seen more often is that as the test goes on the cases that result in the required behaviour get less and less likely. Theoretically, the probabilities that an infinite loop is reached is infinitely small – However it exists, so must be counted as a possibility.

Question 2: What is the shortest possible output for the test, in terms of the number of lines output?

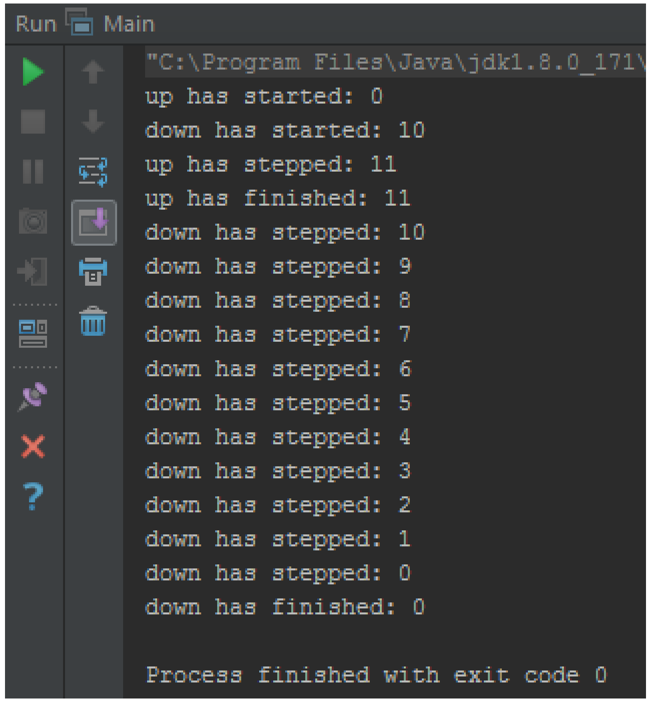
14 Lines.
This is the case in which one counter reaches its goal value before the other counter has stepped once. I discuss this in slightly more detail in the Demo Code section above. This is far more likely in this scenario however, because the counters start at the others' goal value. This means that the only situation where the counters 'compete' with each other is when one of the counters steps before the other can test its goal condition. From then on the counters are competing with each other. It's even possible, however unlikely, that the counter that starts first (being set to its goal value before stepping) could force the other counter all the way back to its goal condition. The more likely (and observed) scenario is the inverse. An example of the described situation occurring:



```
"C:\Program Files\Java\jdk1.8.0_171\bin\java" ...
up has started: 0
up has stepped: 1
up has stepped: 2
up has stepped: 3
up has stepped: 4
up has stepped: 5
up has stepped: 6
up has stepped: 7
up has stepped: 8
up has stepped: 9
up has stepped: 10
up has finished: 10
down has started: 10
down has stepped: 9
down has stepped: 8
down has stepped: 7
down has stepped: 6
down has stepped: 5
down has stepped: 4
down has stepped: 3
down has stepped: 2
down has stepped: 1
down has stepped: 0
down has finished: 0
```

Question 3: What is the largest possible value that the count can reach when the test is run?

11.



```
"C:\Program Files\Java\jdk1.8.0_171\
up has started: 0
down has started: 10
up has stepped: 11
up has finished: 11
down has stepped: 10
down has stepped: 9
down has stepped: 8
down has stepped: 7
down has stepped: 6
down has stepped: 5
down has stepped: 4
down has stepped: 3
down has stepped: 2
down has stepped: 1
down has stepped: 0
down has finished: 0

Process finished with exit code 0
```

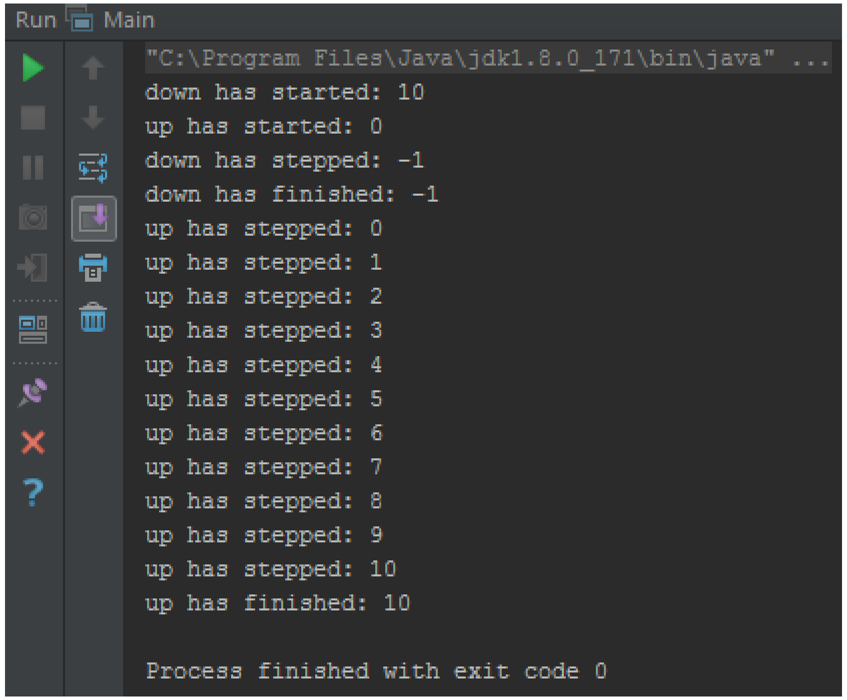
Thread A (Up)	Thread B (Down)	Counter.
Start()	-	-
¬ startCount()	-	-
¬ ¬ counter = from (0)	-	0
¬ isFinished() = false	-	0
¬ stepCount()	-	0
¬ ¬ delay()	-	0
¬ ¬ ^^^^^^^	Start()	0
¬ ¬ ^^^^^^^	¬ startCount()	0
¬ ¬ ^^^^^^^	¬ ¬ counter = from (10)	10
¬ ¬ ^^^^^^^	//Anything can happen here up to and excluding stepCount ¬ counter += step;//	10
¬ ¬ counter += step;	¬ ¬ ^^^^^^^	11
//Process Continues//	//Process Continues//	//Process Continues//

This was an observed situation.

The sequence of events that leads to this occurring are shown above and to the right.

This is a consequence of parallel programming in that when a process starts is unpredictable. Unless checks are made against situations like this, they have the potential to occur and cause unintended side effects. As discussed earlier, this demonstrates how a larger system could experience a fatal error from a read/write error caused by parallel programming.

Question 4: What is the lowest possible value that the count can reach when the test is run?



```
"C:\Program Files\Java\jdk1.8.0_171\bin\java" ...
down has started: 10
up has started: 0
down has stepped: -1
down has finished: -1
up has stepped: 0
up has stepped: 1
up has stepped: 2
up has stepped: 3
up has stepped: 4
up has stepped: 5
up has stepped: 6
up has stepped: 7
up has stepped: 8
up has stepped: 9
up has stepped: 10
up has finished: 10

Process finished with exit code 0
```

-1.

The sequence of events that leads to this situation is exactly the same as the ‘11’ situation however thread B starts before thread A.

Self Evaluation.

For 2 marks I was asked to provide evidence of implementation which I have done so in the (Model) and (Logbook) sections.
For 3 marks I was asked to provide evidence of multiple runs which can be found in the (Logbook) section.
I was also asked to correctly identify min and max values, an answer to which I believe to be correct can be found in (Lobook) -> Question 3/4.
For 4 marks I was asked to produce a clear explanation of the min and max values which I've accomplished using a table of atomic actions (line calls) in (Logbook) -> Question 3/4.
Finally for 5 marks I was asked to provide a good analysis of the termination question. I believe my answers to this week satisfy this in two sections; My analysis in the (Demo code) section, and the actual answer in (Logbook) -> Question 1.

For the reasons stated above, I believe my work this week is worth all 5 marks available.

Practical 10 (Week 14)

Section 1 – Nondeterministic results from concurrent processes.

Question 1 – Rewrite the two parallel process so that there is at most one critical reference in each line of code.

- You may introduce new local variables if necessary. E.g. In process 1 you might define a local temporary variable, p1, say, and assign y+1 to p1, and then assign p1 + 1 to x. Does it make any difference which way you do it?

Code listing.

Process1

```java
package processes;
/**
 * One version of the process --- assign y+1 to x.
 */
public class Process1 extends Process {
    public void run() {
        //read in the variable. 1 reference (read)
        int p = y;

        //calculate and store the new variable. 1 reference (write).
        x = p + 1;
    }
}
```

Process2

```java
package processes;
/**
 * One version of the process --- assign x+1 to y.
 */
public class Process2 extends Process {
    public void run()
    {
        //read in the variable. 1 reference (read)
        int p = x;

        //calculate and store the new variable. 1 reference (write).
        y = p + 1;
    }
  }
```

Does it make any difference which way you do it?

There were two options available to accomplish this. The first was the method discussed in the question, using three lines. The second method was the one I used, with two lines of code.
Reynolds Criterium is satisfied either way we do this, which means any sequence of time slices is possible.

Method 1 – 2 lines.

Possibilities.

```
a = p1=x
b = y=p1+1
1 = p2=y
2 = x=p2+1

(x,p1,y,p2) (0,-,2,-)

a,b,1,2 - (0,0,2,-)_(0,0,1,-)_(0,0,1,1)_"(2,0,1,1)"
a,1,b,2 - (0,0,2,-)_(0,0,2,2)_(0,0,1,2)_"(3,0,1,2)"
a,1,2,b - (0,0,2,-)_(0,0,2,2)_(3,0,2,2)_"(3,0,4,2)"

1,a,b,2 - (0,-,2,2)_(0,0,2,2)_(0,0,1,2)_"(3,0,1,2)"
1,a,2,b - (0,-,2,2)_(0,0,2,2)_(3,0,2,2)_"(3,0,1,2)"
1,2,a,b - (0,-,2,2)_(3,-,2,2)_(3,3,2,2)_"(3,3,4,2)"
```

Results.

X	Y	Frequency	Probability
2	1	1	1/6
3	1	3	½
3	4	2	1/3

Method 2 – 3 lines.

Possibilites.

```
a = p1=x
b = p1++
c = y=p1
1 = p2=y
2 = p2++
3 = x=p2

(x,p1,y,p2) (0,-,2,-)

a,b,c,1,2,3 - (0,0,2,-)_(0,1,2,-)_(0,1,1,-)_(0,1,1,1)_(0,1,1,2)_"(2,1,1,2)"
a,b,1,c,2,3 - (0,0,2,-)_(0,1,2,-)_(0,1,2,2)_(0,1,1,2)_(0,1,1,3)_"(3,1,1,3)"
a,b,1,2,c,3 - (0,0,2,-)_(0,1,2,-)_(0,1,2,2)_(0,1,2,3)_(0,1,1,3)_"(3,1,1,3)"
a,b,1,2,3,c - (0,-,2,-)_(0,1,2,-)_(0,1,2,2)_(0,1,2,3)_(3,1,2,3)_"(3,1,1,3)"

a,1,b,c,2,3 - (0,0,2,-)_(0,0,2,2)_(0,1,2,2)_(0,1,1,2)_(0,1,1,3)_"(3,1,1,3)"
a,1,b,2,c,3 - (0,0,2,-)_(0,0,2,2)_(0,1,2,2)_(0,1,2,3)_(0,1,1,3)_"(3,1,1,3)"
a,1,b,2,3,c - (0,0,2,-)_(0,0,2,2)_(0,1,2,2)_(0,1,2,3)_(3,1,2,3)_"(3,1,1,3)"
a,1,2,3,b,c - (0,0,2,-)_(0,0,2,2)_(0,0,2,3)_(3,0,2,3)_(3,1,2,3)_"(3,1,1,3)"
a,1,2,b,3,c - (0,0,2,-)_(0,0,2,2)_(0,0,2,3)_(0,1,2,3)_(3,1,2,3)_"(3,1,1,3)"
a,1,2,b,c,3 - (0,0,2,-)_(0,0,2,2)_(0,0,2,3)_(0,1,2,3)_(0,1,1,3)_"(3,1,1,3)"

1,2,3,a,b,c - (0,-,2,2)_(0,-,2,3)_(3,-,2,3)_(3,3,2,3)_(3,4,2,3)_"(3,4,4,3)"
1,2,a,3,b,c - (0,-,2,2)_(0,-,2,3)_(0,0,2,3)_(3,0,2,3)_(3,1,2,3)_"(3,1,1,3)"
1,2,a,b,3,c - (0,-,2,2)_(0,-,2,3)_(0,0,2,3)_(0,1,2,3)_(3,1,2,3)_"(3,1,1,3)"
1,2,a,b,c,3 - (0,-,2,2)_(0,-,2,3)_(0,0,2,3)_(0,1,2,3)_(0,1,1,3)_"(3,1,1,3)"

1,a,2,3,b,c - (0,-,2,2)_(0,0,2,2)_(0,0,2,3)_(3,0,2,3)_(3,1,2,3)_"(3,1,1,3)"
1,a,2,b,3,c - (0,-,2,2)_(0,0,2,2)_(0,0,2,3)_(0,1,2,3)_(3,1,2,3)_"(3,1,1,3)"
1,a,2,b,c,3 - (0,-,2,2)_(0,0,2,2)_(0,0,2,3)_(0,1,2,3)_(0,1,1,3)_"(3,1,1,3)"
1,a,b,c,2,3 - (0,-,2,2)_(0,0,2,2)_(0,1,2,2)_(0,1,1,2)_(0,1,1,3)_"(3,1,1,3)"
1,a,b,2,c,3 - (0,-,2,2)_(0,0,2,2)_(0,1,2,2)_(0,1,2,3)_(0,1,1,3)_"(3,1,1,3)"
1,a,b,2,3,c - (0,-,2,2)_(0,0,2,2)_(0,1,2,2)_(0,1,2,3)_(3,1,2,3)_"(3,1,1,3)"
```

Results.

X	Y	Frequency	Probability
2	1	1	1/20
3	1	18	9/10
3	4	1	1/20

Conclusion

The implementation method absolutely makes a difference. Using method 2 (3 lines) makes it far more likely that the (presumed) correct output will occur. Comparing the two shows 0.9 : 0.5 probability.

NOTE: My results assume each action has an equal probability of being executed. This is not the case in real hardware, and these are merely theoretical answer. The real probabilities are likely much closer than the concluded ones here.

Question 2 – Using your code as a model for the implementation of the processes and assuming that Reynold’s criterium is met, what are the possible values for x and y upon termination of this programme?

I.e. assuming that each line of your code can be considered as an atomic action (cannot be interrupted), consider all the possible interleavings of the lines of code in the rewritten processes, and trace the values of the variables through these interleavings.

Answers to all this can be found in the above ‘Question 2’.

Question 3 – Reynold's criterium may not be met.

Assume that this programme is implemented on some one address machine, with x = y + 1 implemented as:

- LDA :y:
- ADD &1
- STA :x:

And y = x + 1 is implemented similarly.

Show how this programme can terminate with x having the value 0 and y having the value 1.

Possibilites.

```
a = LDA y
b = ADD 1
c = STA x
1 = LDA x
2 = ADD 1
3 = STA y

(x,y,ac) - (0,2,-)

a,b,c,1,2,3 - (0,2,2)_(0,2,3)_(3,2,3)_(3,2,3)_(3,2,4)_"(3,4,4)"
a,b,1,c,2,3 - (0,2,2)_(0,2,3)_(0,2,0)_(0,2,0)_(0,2,1)_"(0,1,1)"
a,b,1,2,c,3 - (0,2,2)_(0,2,3)_(0,2,0)_(0,2,1)_(1,2,1)_"(1,1,1)"
a,b,1,2,3,c - (0,2,2)_(0,2,3)_(0,2,0)_(0,2,1)_(0,1,1)_"(1,1,1)"

a,1,b,2,3,c - (0,2,2)_(0,2,0)_(0,2,1)_(0,2,2)_(0,2,2)_"(2,2,2)"
a,1,b,2,c,3 - (0,2,2)_(0,2,0)_(0,2,1)_(0,2,2)_(2,2,2)_"(2,2,2)"
a,1,b,c,2,3 - (0,2,2)_(0,2,0)_(0,2,1)_(1,2,1)_(1,2,2)_"(1,2,2)"
a,1,2,3,b,c - (0,2,2)_(0,2,0)_(0,2,1)_(0,1,1)_(0,1,2)_"(2,1,2)"
a,1,2,b,3,c - (0,2,2)_(0,2,0)_(0,2,1)_(0,2,2)_(0,2,2)_"(2,2,2)"
a,1,2,b,c,3 - (0,2,2)_(0,2,0)_(0,2,1)_(0,2,2)_(2,2,2)_"(2,2,2)"

1,2,3,a,b,c - (0,2,0)_(0,2,1)_(0,1,1)_(0,1,1)_(0,1,2)_"(2,1,2)"
1,2,a,3,b,c - (0,2,0)_(0,2,1)_(0,2,2)_(0,2,2)_(0,2,3)_"(3,2,3)"
1,2,a,b,3,c - (0,2,0)_(0,2,1)_(0,2,2)_(0,2,3)_(0,3,3)_"(3,3,3)"
```

```
1,2,a,b,c,3 - (0,2,0)_(0,2,1)_(0,2,2)_(0,2,3)_(3,3,3)_"(3,3,3)"

1,a,2,b,c,3 - (0,2,0)_(0,2,2)_(0,2,3)_(0,2,4)_(4,2,4)_"(4,4,4)"
1,a,2,b,3,c - (0,2,0)_(0,2,2)_(0,2,3)_(0,2,4)_(4,4,4)_"(4,4,4)"
1,a,2,3,b,c - (0,2,0)_(0,2,2)_(0,2,3)_(0,3,3)_(0,3,4)_"(4,3,4)"
1,a,b,c,2,3 - (0,2,0)_(0,2,2)_(0,2,3)_(3,2,3)_(3,2,4)_"(3,4,4)"
1,a,b,2,c,3 - (0,2,0)_(0,2,2)_(0,2,3)_(0,2,4)_(4,2,4)_"(4,4,4)"
1,a,b,2,3,c - (0,2,0)_(0,2,2)_(0,2,3)_(0,2,4)_(0,4,4)_"(4,4,4)"
```

Results

X	Y	Frequency	Probability
0	1	1	0.05
1	1	2	0.1
1	2	1	0.05
2	1	2	0.1
2	2	4	0.2
3	2	1	0.05
3	3	2	0.1
3	4	2	0.1
4	3	1	0.05
4	4	4	0.2

Conclusion.

As can be shown by the above possibilities and results, there is a sequence of atomic actions here that result in the x,y values of 0,1. This sequence is highlighted.

Why does this implementation not satisfy Reynold's Criterium?

This implementation does not satisfy the criterium because when a store operation is performed this is two critical references. The variable is read from the acc and written into the mem address in one go. This is two references so the criterium breaks down.
Furthermore, if it's assumed these instructions **do** only have one critical reference each, then the criterium is still not met because the result is not deterministic. Because it's using the shared acc, the result of accessing this acc is not deterministic.

Section 2 – Ferrcarriles de America del sur

Method 1

They put a large basket at the entrance to the pass. Before entering the pass the engine driver must stop, walk to the basket and feel if there is a stone in the basket. If the basket is empty the engine driver finds a stone and puts it in the basket in order to indicate that he is about to enter the pass. A sson as he has crossed the pass he walks back and retrieves his stone in order to indicate that the pass is free. Finally he walks back to his train and continues his journey.

If one of the engine drivers finds a stone already in the basket he leaves it in there, and has a siesta. When he wakes up again he repeats the actions he took when he arrived.

There were soon problems with this system.

Problem 1.

The Bolivians complained that subversive timetables put together by the Peruvian railways could block the Bolivian trains for ever. Give a scenario, using the code, that shows how this could happen.

Code.

```
public void runTrain() throws RailwaySystemError
{
        Clock clock = getRailwaySystem().getClock();
        while (!clock.timeOut() && !errorFlag)
        {
                choochoo();
                while (getSharedBasket().hasStone(this))
                {
                        siesta();
                }
                getSharedBasket().putStone(this);
                crossPass();
                getSharedBasket().takeStone(this);
        }
}
```

Scenario

Peru	Bolivia	Basket
RunTrain()	-	-
.Clock clock	-	-
..while(!clock...) (false)	-	-
...choochoo()	-	-
...while(getBasket()...) (false)	-	-
...putStone()	-	Peru
...crossPass()	-	Peru
^	RunTrain()	Peru
|	.Clock clock	Peru

|	..while(!clock...) (false)	Peru
|	...choochoo()	Peru
|	...while(getBasket()...) (true)	Peru
|	....siesta()	Peru
...takeStone()	^	-
..while(!clock...) (false)	|	-
...choochoo()	|	-
...while(getBasket()...) (false)	|	-
...putStone()	|	Peru
...crossPass()	|	Peru
^	...while(getBasket()...) (true)	Peru
|	....siesta()	Peru
Etc.	Etc.	Etc.
Etc.	Etc.	Etc.

Problem 2.

More seriously, one day the trains crashed. Give a scenario, using the code, that shows how this could happen.

Scenario

Peru	Bolivia	Basket
RunTrain()	-	-
.Clock clock	-	-
..while(!clock...) (false)	-	-
...choochoo()	-	-
...while(getSharedBasket()...)	-	-
^	RunTrain()	-
|	.Clock clock	-
|	..while(!clock...) (false)	-
|	...choochoo()	-
|	...while(getSharedBasket...()	-
...while(...hasStone()) (false)	^	-
^	...while(...hasStone()) (false)	-
...putStone()	^	Peru
^	...putStone()	Bolivia
crossPass()	^	Bolivia
^	crossPass()	Bolivia
CRASH	CRASH	Bolivia

Method 2.

After the crash the train drivers agreed on a different system. The Peruvian engine driver would wait by the entrance to the pass until the basket is empty, cross the pass, and then walk back to put a stone in the basket. The Bolivian engine driver would wait until there is a stone in the basket, then cross the pass, and then walk back to take the stone out of the basket.

Provide code for the Peru and Bolivia classes for this solution.

Peru

```
public void runTrain() throws RailwaySystemError {
   Clock clock = getRailwaySystem().getClock();
   while (!clock.timeOut())
   {
      //move to start of pass.
      choochoo();

      //wait until basket empty.
      while(getSharedBasket().hasStone(this) || getSharedBasket().hasStone(getRailwaySystem().getNextRailway(this)))
      {
          siesta();
      }

      //cross the pass.
      crossPass();
      //walk back to put a stone in the basket.
      getSharedBasket().putStone(this);
   }
}
```

Bolivia

```
public void runTrain() throws RailwaySystemError {
   Clock clock = getRailwaySystem().getClock();
   while (!clock.timeOut())
   {
      //move to start of pass.
      choochoo();

      //wait until stone in basket.
      while(!getSharedBasket().hasStone(this) || !getSharedBasket().hasStone(getRailwaySystem().getNextRailway(this)))
      {
         siesta();
      }

      //cross the pass.
      crossPass();
      //walk back to take the stone out of the basket.
      getSharedBasket().takeStone(this);
   }
}
```

Does this method prevent crashes.

Yes. The train companies take it in turns to cross the pass hence there can never be a collision.

There was immediately a dispute between Peru and Bolivia over timetabling, explain why.

This method means that unless both train systems have interweaved timetables, there will be trains waiting around at the start of the pass. If Peru wants to increase the frequency of its trains, then they would be waiting a long time for the less frequent Bolivian trains to cross the pass. This is inefficient and restrictive to both Railway systems.

Method 3.

The train companies agreed that they needed expert help, and hired a consultant, a graduate of a Russell University. He proposed using two baskets, one for each engine driver. If an engine driver came to the entrance to the pass he must first check the other driver's basket to see if it is empty. If it is he puts a stone in his own basket, crosses the pass and then walks back to remove the stone from his basket.

If there is a stone in the other driver's basket, he has a siesta, then checks the other driver's basket again, repeating this process until the other driver's basket is empty. When it is he puts a stone in his own basket, crosses the pass and then walks back to remove the stone from his basket.

Unfortunately, the trains now crashed again, explain how this can happen.

Code

```
public void runTrain() throws RailwaySystemError {
   Clock clock = getRailwaySystem().getClock();
   while (!clock.timeOut())
   {
      //move to start of pass.
      choochoo();

      //Find out what the other railway is
      Railway otherRail = getRailwaySystem().getNextRailway(this);

      //if the other basket has a stone in
      while(otherRail.getBasket().hasStone(otherRail))
      {
         //take a siesta.
         siesta();
      }

      //Put a stone in his own basket.
      getBasket().putStone(this);

      //cross the pass
      crossPass();

      //remove the stone from your own basket.
      getBasket().takeStone(this);

   }
}
```

Scenario

Peru	**Bolivia**	**Peru Basket**	**Bolivia Basket**
RunTrain()	-	-	-
.Clock clock	-	-	-
..choochoo()	-	-	-
..Railway otherRail	-	-	-
..while(otherRail.getBasket()...) (Bolivia)	-	-	-
^	RunTrain()	-	-
|	.Clock clock	-	-
|	..choochoo()	-	-
|	..Railway otherRail	-	-
|	..while(otherRail.getBasket()...) (Peru)	-	-
..while(...hasStone()) (false)	^	-	-
^	..while(...hasStone()) (false)	-	-
..putStone(this)	^	Y	-
^	..putStone(this)	Y	Y
..crossPass()	^	Y	Y
^	..crossPass()	Y	Y
CRASH	CRASH	Y	Y

Method 4.

They went back to the consultant, who told them to put stones in their own baskets before checking the other driver's basket.

Give code for this attempted solution.

```
public void runTrain() throws RailwaySystemError {
   Clock clock = getRailwaySystem().getClock();
   while (!clock.timeOut())
{
    //move to start of pass.
      choochoo();

      //Find out what the other railway is
      Railway otherRail = getRailwaySystem().getNextRailway(this);

          //Put a stone in his own basket.
          getBasket().putStone(this);

      //if the other basket has a stone in
          while(otherRail.getBasket().hasStone(otherRail))
          {
              //take a siesta.
              siesta();
          }

          //cross the pass
          crossPass();

          //remove the stone from your own basket.
          getBasket().takeStone(this);

   }
  }
```

One day the trains stopped running. Explain how this can happen.

Peru	Bolivia	Peru Basket	Bolivia Basket
`RunTrain()`	-	-	-
`.Clock clock`	-	-	-
`..choochoo()`	-	-	-
`..Railway otherRail`	-	-	-
`..putStone(this)`	-	Y	-
`..while(otherRail.getBasket()...)`	-	Y	-
^	`RunTrain()`	Y	-
|	`.Clock clock`	Y	-
|	`..choochoo()`	Y	-
|	`..Railway otherRail`	Y	-
|	`..putStone(this)`	Y	Y
|	`..while(otherRail.getBasket()...)`	Y	Y
`..while(...hasStone(otherRail)) (true)`	^	Y	Y
`...siesta()`	|	Y	Y
^	`..while(...hasStone(otherRail()) (true)`	Y	Y
|	`...siesta()`	Y	Y
Etc.	Etc.	Y	Y
Etc.	Etc.	Y	Y

(Model) Method 5.

The consultant was asked for advice again which, for a fat fee, he provided. He told the drivers that, of course, before having a siesta they should remove the stone from their basket, and put it back in again as soon as they wake up.

Provide code for this solution.

```java
public void runTrain() throws RailwaySystemError {
   Clock clock = getRailwaySystem().getClock();
   while (!clock.timeOut())
   {
      //move to start of pass.
      choochoo();

      //Find out what the other railway is
      Railway otherRail = getRailwaySystem().getNextRailway(this);

      //Put a stone in his own basket.
      getBasket().putStone(this);

      //if the other basket has a stone in
      while(otherRail.getBasket().hasStone(otherRail))
      {
         //Remove the stone before having a siesta.
         getBasket().takeStone(this);

         //take a siesta.
         siesta();

         //take the stone out immediately after.
         getBasket().putStone(this);
      }

      //cross the pass
      crossPass();

      //remove the stone from your own basket.
      getBasket().takeStone(this);

   }
}
```

This was an improvement on the previous attempts but, after some experience with it, the railways were still not happy. Why not?
Situations keep occurring where the railways were waking up, and checking the other basket just before the other driver takes his stone out. This means that both drivers end up checking the baskets and taking indefinite siestas. The result of this is that no trains go through the pass at all.

Provide a scenario to demonstrate your answer to question 5b.

Peru	Bolivia	Peru Basket	Bolivia Basket
`RunTrain()`	-	-	-
`.choochoo()`	-	-	-
`.Railway otherRail`	-	-	-
`.putStone(this)`	-	Y	-
`.while(otherRail.getBasket()...)`	-	Y	-
^	`RunTrain()`	Y	-
|	`.choochoo()`	Y	-
|	`.Railway otherRail`	Y	-
|	`.putStone(this)`	Y	Y
`.while(...hasStone(otherRail)) (true)`	^	Y	Y
^	`.while(...hasStone(otherRail)) (true)`	Y	Y
`..takeStone(this)`	^	-	Y
`..siesta()`	|	-	Y
^	`..takeStone(this)`	-	-
|	`..siesta()`	-	-
`..putStone(this)`	^	Y	-
^	`..putStone(this)`	Y	Y
`.while(...) (true)`	^	Y	Y
^	`.while(...) (true)`	Y	Y
Etc.	Etc.	Etc.	Etc.

(Logbook) Method 6.

The train companies sacked the consultant, and hired a recent Huddersfield graduate. She immediately saw the solution, having paid careful attention to Hugh’s lecture on Dekker’s algorithm. Implement her solution.

Code.

```
public void runTrain() throws RailwaySystemError {
   Clock clock = getRailwaySystem().getClock();
   while (!clock.timeOut())
{
    //move to start of pass.
      choochoo();

      //Find out what the other railway is
      Railway otherRail = getRailwaySystem().getNextRailway(this);

          //if it's no-one's turn, it is Peru's turn.
          if(!getSharedBasket().hasStone(this) && !getSharedBasket().hasStone(otherRail))
          {
              getSharedBasket().putStone(this);
          }

          //Put a 'permission-request stone' in his own basket.
          getBasket().putStone(this);

      //if the other basket has a stone in (other rail is requesting permission)
          while(otherRail.getBasket().hasStone(otherRail))
          {
           //if it's not peru's turn as designated by the shared basket.
              if(getSharedBasket().hasStone(otherRail))
              {
                  //Remove your 'permission-request stone' before having a siesta.
                  getBasket().takeStone(this);

                  //take a siesta until it's this train's turn.
                  while(!getSharedBasket().hasStone(this))
                  {
                      siesta();
                  }

                  //put your 'permission-request stone' back in your own basket.
                  getBasket().putStone(this);
              }
          }

          //cross the pass
          crossPass();
          //It is now the other rail's turn
          getSharedBasket().takeStone(this);
          getSharedBasket().putStone(otherRail);

          //remove the 'permission-request stone' from your own basket.
          getBasket().takeStone(this);

   }
  }
```

Self Evaulation.

For 3 marks, I was asked to provide a Mostly correct implementation using train code. Extending this to 4 marks required a correct implementation.

My implementation works as it should and provides a faithful recreation of Dekker’s algorithm whilst pertaining to the ‘train-code’ conventions established in the template.

For a full 5 marks, I was asked to provide a detailed analysis of code. Throughout both section 1 and 2 I provide detailed analysis’ and walkthroughs of scenarios in concurrent critical section code.

For the reasons stated above, I believe my work is worth the full 5/5 marks available this week.

Practical 11 (Week 15)

(Logbook) Questions

Programme arguments

- **Buffer size**
    - Positive, non zero, integer.
- **Run time**
    - Positive Integer/Decimal.
    - Specifies the seconds the buffer system should run for.
- **Producer delay**
    - Positive Integer/Decial.
        - OR slow (2.0) / medium (1.0) / fast (0.5)
    - Producer pauses between each buffer access for a random time between zero and this value.
- **Consumer delay**
    - Same as producer delay, but for the consumer.

Question 1

The order of the *'criticalSection.P()'* and *'noOfElements.P()'* in the *'Buffer'* class's *'get()'* method is essential.

However the other of the *'vote()s'* in the same class's *'put()'* method are not.

Identify the corresponding piece of code in the 'Buffer' class and make the change.

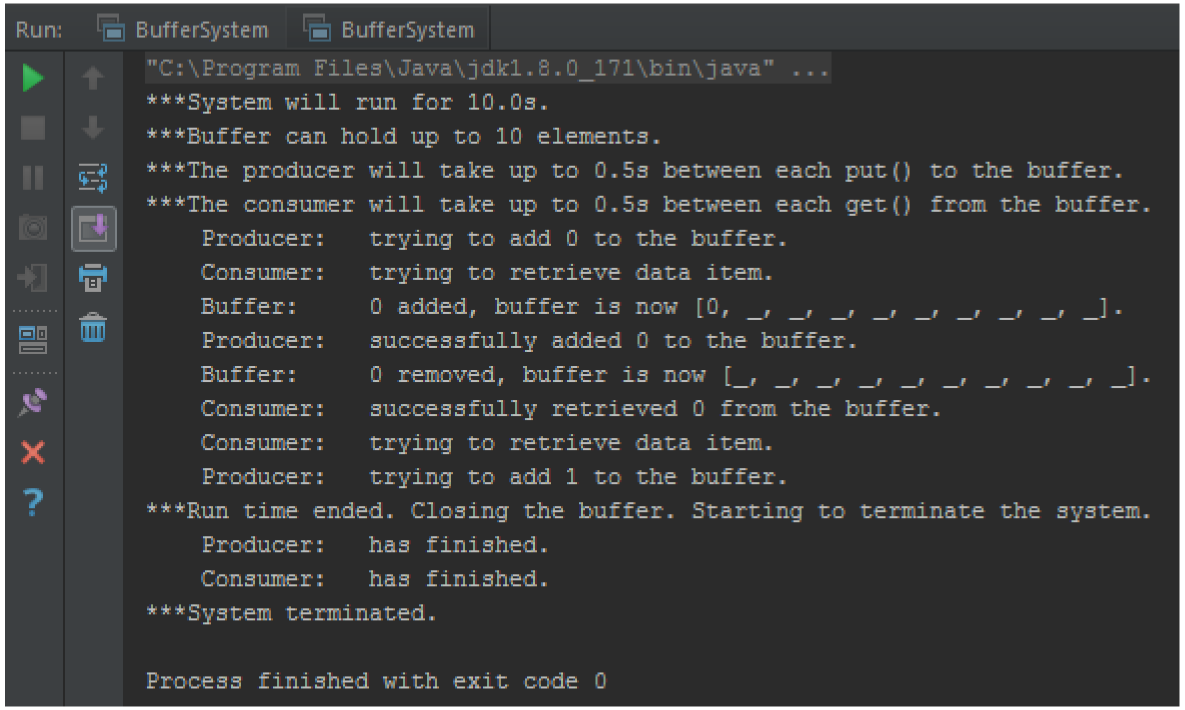
Previous code.

```
public T get() throws BufferError, SemaphoreLimitError {
    T item;
    try {
        noOfElements.poll(); // is there at least one data item in the buffer?
        criticalSection.poll();   // is the buffer available?
        item = getItem(); // add the data item
        criticalSection.vote();   // make the buffer available again
        noOfSpaces.vote();  // there is now one more space in the buffer
    } catch (InterruptedException ie) {
        throw new BufferError("Buffer: Data item could not be retrieved from the buffer.\n" +
                              "\t" + ie.getMessage());
    }
    return item;
}
```

New code.

```
public T get() throws BufferError, SemaphoreLimitError {
    T item;
    try {
        criticalSection.poll();   // is the buffer available?
        noOfElements.poll(); // is there at least one data item in the buffer?
        item = getItem(); // add the data item
        criticalSection.vote();   // make the buffer available again
        noOfSpaces.vote();  // there is now one more space in the buffer
    } catch (InterruptedException ie) {
        throw new BufferError("Buffer: Data item could not be retrieved from the buffer.\n" +
                              "\t" + ie.getMessage());
    }
    return item;
}
```

Can you produce an error situation?



```
"C:\Program Files\Java\jdk1.8.0_171\bin\java" ...
***System will run for 10.0s.
***Buffer can hold up to 10 elements.
***The producer will take up to 0.5s between each put() to the buffer.
***The consumer will take up to 0.5s between each get() from the buffer.
    Producer:   trying to add 0 to the buffer.
    Consumer:   trying to retrieve data item.
    Buffer:     0 added, buffer is now [0, _, _, _, _, _, _, _, _, _].
    Producer:   successfully added 0 to the buffer.
    Buffer:     0 removed, buffer is now [_, _, _, _, _, _, _, _, _, _].
    Consumer:   successfully retrieved 0 from the buffer.
    Consumer:   trying to retrieve data item.
    Producer:   trying to add 1 to the buffer.
***Run time ended. Closing the buffer. Starting to terminate the system.
    Producer:   has finished.
    Consumer:   has finished.
***System terminated.

Process finished with exit code 0
```

As can be observed within this screen shot. The consumer and the producer reach a state of deadlock between each other.

The consumer (in the new method), checks if the critical section is empty before considering whether or not there is an element held in the buffer or not. These are bounded semaphores, so whilst the consumer is allowed past the critical section semaphore, when it gets to the next line, polling the number of elements, it cannot get past as there are no elements in the buffer. This means the consumer is told to wait for the producer.

Meanwhile, the producer is attempting to access the critical section to put an element in the buffer. It of course cannot do this, as the consumer has already stated it is 'within' the critical section. Therefore the producer is waiting for the consumer to get out of the critical section.

It is plainly obvious that neither the producer/consumer can do anything at this point, which results in the state of deadlock observed.

Question 2.

Why does the error situation arise when the code is changed as described in question1?

As can be observed within the screen shot above, the consumer and the producer reach a state of deadlock between each other.

The consumer (in the new method), checks if the critical section is empty before considering whether or not there is an element held in the buffer or not. These are bounded semaphores, so whilst the consumer is allowed past the critical section semaphore, when it gets to the next line, polling the number of elements, it cannot get past as there are no elements in the buffer. This means the consumer is told to wait for the producer.

Meanwhile, the producer is attempting to access the critical section to put an element in the buffer. It of course cannot do this, as the consumer has already stated it is 'within' the critical section. Therefore the producer is waiting for the consumer to get out of the critical section.

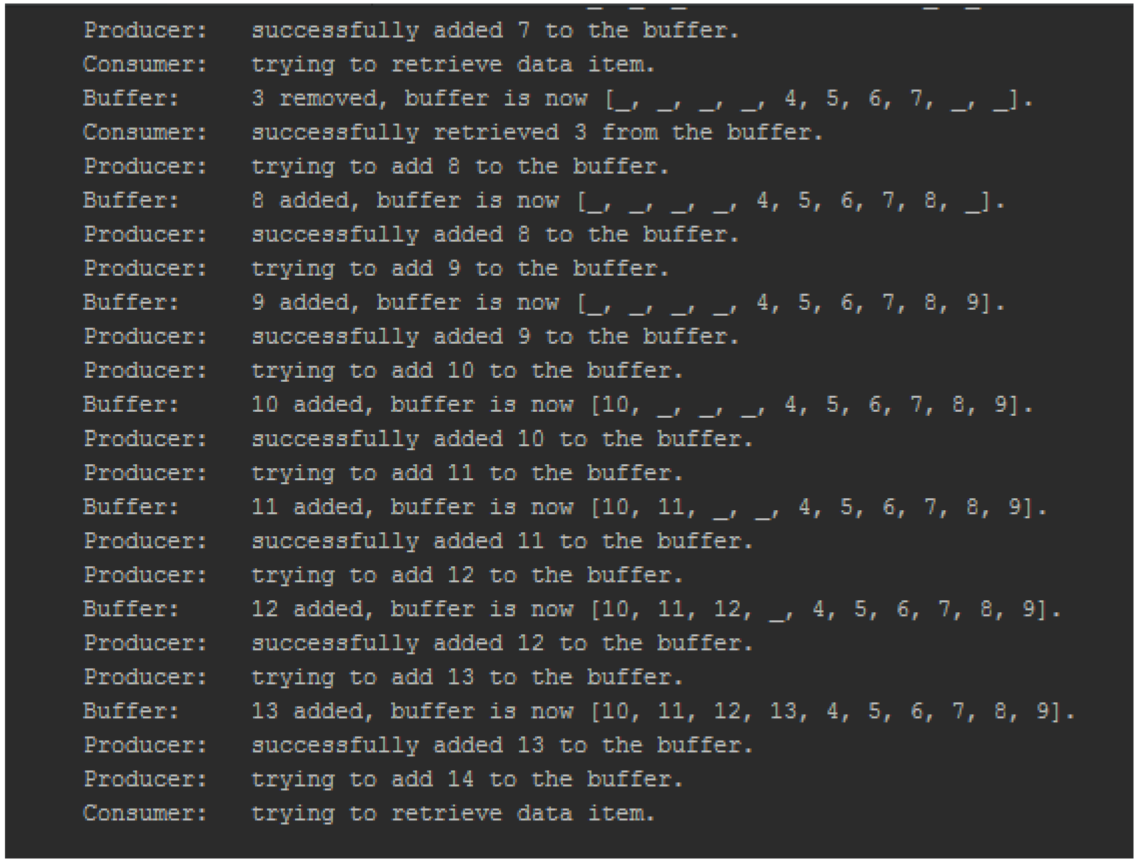
It is plainly obvious that neither the producer/consumer can do anything at this point, which results in the state of deadlock observed.

Why does it not arise in the original code?

In the original code, the consumer must first know that there is an element within the buffer to take before it is allowed to request permission to access the critical section. Essentially, this means that the producer always has an opportunity to put an element in the buffer, and that the consumer will never be allowed to enter the critical section when the buffer contains no elements.

Question 3.

Is the order of the calls of P() in the Buffer *class's* put() *method also essential?*



```
Producer:   successfully added 7 to the buffer.
Consumer:   trying to retrieve data item.
Buffer:     3 removed, buffer is now [_, _, _, _, 4, 5, 6, 7, _, _].
Consumer:   successfully retrieved 3 from the buffer.
Producer:   trying to add 8 to the buffer.
Buffer:     8 added, buffer is now [_, _, _, _, 4, 5, 6, 7, 8, _].
Producer:   successfully added 8 to the buffer.
Producer:   trying to add 9 to the buffer.
Buffer:     9 added, buffer is now [_, _, _, _, 4, 5, 6, 7, 8, 9].
Producer:   successfully added 9 to the buffer.
Producer:   trying to add 10 to the buffer.
Buffer:     10 added, buffer is now [10, _, _, _, 4, 5, 6, 7, 8, 9].
Producer:   successfully added 10 to the buffer.
Producer:   trying to add 11 to the buffer.
Buffer:     11 added, buffer is now [10, 11, _, _, 4, 5, 6, 7, 8, 9].
Producer:   successfully added 11 to the buffer.
Producer:   trying to add 12 to the buffer.
Buffer:     12 added, buffer is now [10, 11, 12, _, 4, 5, 6, 7, 8, 9].
Producer:   successfully added 12 to the buffer.
Producer:   trying to add 13 to the buffer.
Buffer:     13 added, buffer is now [10, 11, 12, 13, 4, 5, 6, 7, 8, 9].
Producer:   successfully added 13 to the buffer.
Producer:   trying to add 14 to the buffer.
Consumer:   trying to retrieve data item.
```

In exactly the same vein as Question 2, this produces exactly the same error with one minor alteration. The producer attempts to put an element into a buffer that is already full before the consumer has a chance to take an element out.

Self evaluation

For 3/5 marks this week, I was asked to recognize and identify the deadlock situation and provide an explanation to its occurrence. I have done both of these.

For 5/5 marks I was asked to provide a detailed explanation for both cases. I believe that my explanation to the get() method's case is sufficient that describing the put() as the get() inverse is a sufficient analysis.

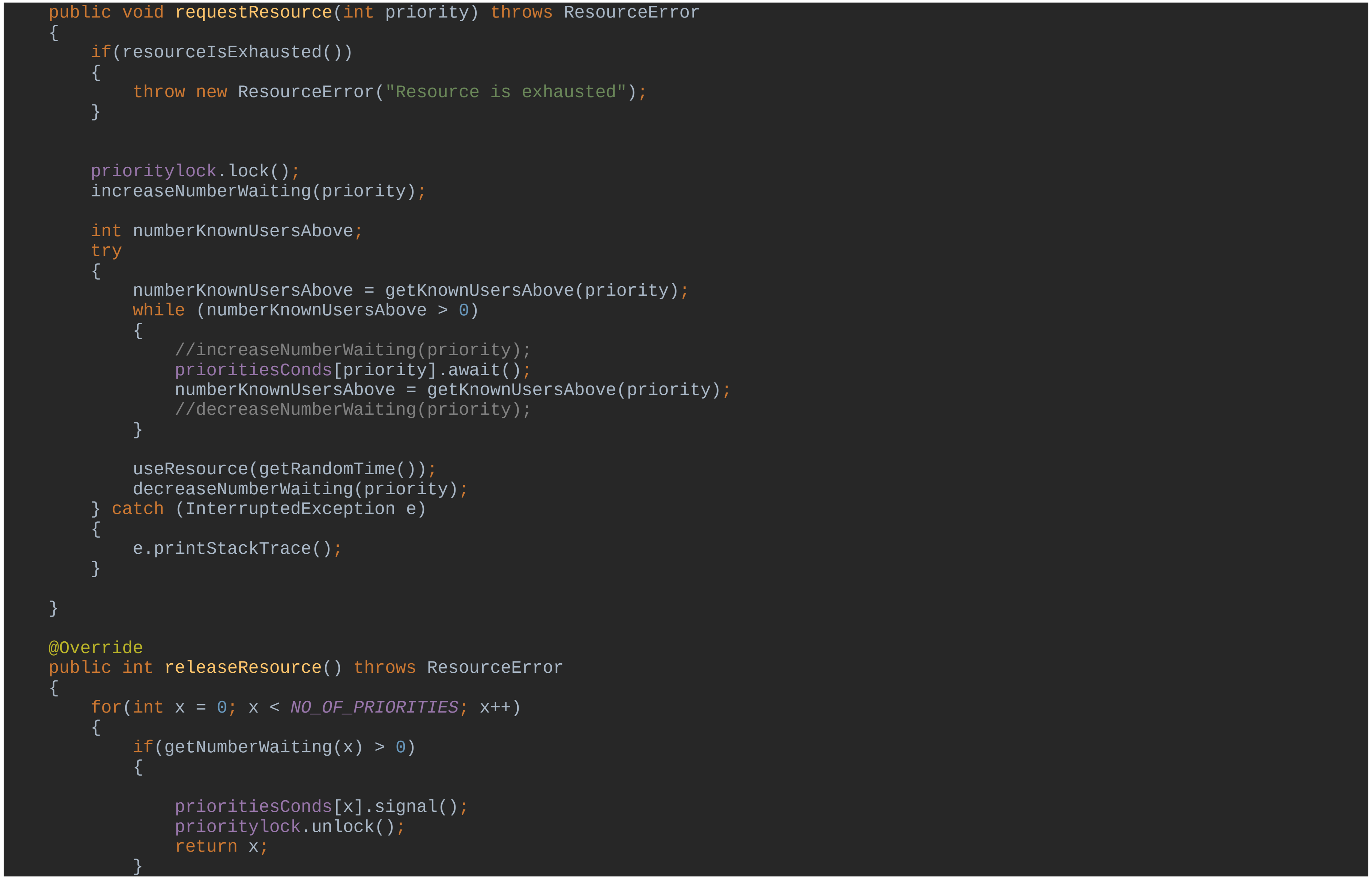
For the reasons stated above, I believe my work is worthy of 5/5 marks this week.

Practical 12 (Week 16)

(Logbook) Question 1.

Code

```java
package resourceManager;

import java.util.Random;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * Created by u1661665(Joshua Pritchard) on 26/02/2019.
 *
 * Version: 26.02.2019
 */
public class LockResourceManager extends BasicResourceManager
{
    final Lock prioritylock;
    final Condition[] prioritiesConds = new Condition[11];


    /**
     * Set the resource and initialise the numbers of waiting processes, and the number of users, to zero.
     *
     * @param resource the resource managed by this manager
     * @param maxUses  the maximum number of uses permitted for this manager's resource.
     */
    public LockResourceManager(Resource resource, int maxUses)
    {
        super(resource, maxUses);
        prioritylock = new ReentrantLock();
        for(int x = 0; x < NO_OF_PRIORITIES; x++)
        {
            prioritiesConds[x] = prioritylock.newCondition();
        }
    }

    @Override
```



```java
    public void requestResource(int priority) throws ResourceError
    {
        if(resourceIsExhausted())
        {
            throw new ResourceError("Resource is exhausted");
        }


        prioritylock.lock();
        increaseNumberWaiting(priority);

        int numberKnownUsersAbove;
        try
        {
            numberKnownUsersAbove = getKnownUsersAbove(priority);
            while (numberKnownUsersAbove > 0)
            {
                //increaseNumberWaiting(priority);
                prioritiesConds[priority].await();
                numberKnownUsersAbove = getKnownUsersAbove(priority);
                //decreaseNumberWaiting(priority);
            }

            useResource(getRandomTime());
            decreaseNumberWaiting(priority);
        } catch (InterruptedException e)
        {
            e.printStackTrace();
        }

    }

    @Override
    public int releaseResource() throws ResourceError
    {
        for(int x = 0; x < NO_OF_PRIORITIES; x++)
        {
            if(getNumberWaiting(x) > 0)
            {

                prioritiesConds[x].signal();
                prioritylock.unlock();
                return x;
            }
        }

        prioritylock.unlock();
        return NONE_WAITING;
    }

    private int getRandomTime()
    {
        Random randTime = new Random();
        return randTime.nextInt(1);
    }

    private int getKnownUsersAbove(int priority)
    {

        int total = 0;
        for(int x = 0; x < priority - 1; x++)
        {
            total += getNumberWaiting(x);
        }


        return total;
    }
}
```

Self evaluation

For 3/5 Marks this week, I was asked to create a (near) working solution with some shortcomings. I believe I have achieved this level as I noticed in testing my implementation does not run the ‘releasing to x priority’ statements. I believe this means my system deals with priorities slightly incorrectly. However, my implementation assigns processes control and allows the resource to be used.

For 4/5 marks, I was asked for a correct implementation with some shortcomings. I believe my process definitely has shortcomings, as I do not have documentation or full testing.

For the reasons discussed above, I believe my work is worth 3/5 marks this week.