ASSIGNMENT 1 SOLUTIONS

Green

1.When the increment function is **not** synchronized these are the samples of the values that I obtain when I run the code:

a) 19870969
b) 19897579
c) 19714079
d) 19869376
e) 19864200

You notice from this trend that even though the final values are close to the expected value, which is 20000000, none of them reach this expected value.

2.a) The values of 200 that are obtained when the counts value is reduced to 100, are generated because each thread is able to obtain the shared field 'count' and increment it by the number of times suggested by the loop before that thread's process is interrupted by the other thread. Because the value of the counts variable is smaller now, each thread is able to finish its task quickly before it is interrupted.

b) In the current state of the software, I would guarantee that the software would always produce 200. This is because given the current counts value, the threads would never interrupt each other.

3. It would not make any difference to use the form count++ or count += 1. This is because both operators increment the value of count by 1 and using either of them is still going to generate an output value of 200(that is if the value of 'counts' in the loop calling the increment function is 100).

The only difference between count++ and count+=1 is that, when count++ is used, the java virtual machine directly increments the value of count. However, when count+=1 is used, it is first converted into count = count + 1 by the compiler and then it is executed.

Yellow

4.After decompiling the code, I can say that the results of my decomposition support my earlier explanation.

5.The **expected** final value after both threads have finished executing is zero. However, after both threads have finished executing, if they are **not** synchronized, the following are samples of the final values that are obtained:

a) -306760
b) -1492483
c) 173265

The methods **have to be** synchronized to obtain the expected value. This is because, when the methods **are not** synchronized, one thread can interrupt another thread at any point in time to execute on an object instance. When this happens, the expected final value would not be

obtained. However, when the methods **are** synchronized, only one thread can execute on an object instance, at a point in time, when that method is called. This allows a given thread to completely finish its process before it is interrupted by another thread, hence giving the expected final value.

6. In the case where:
i) Both of the increment and decrement methods are not synchronized, these are some of the results:

a) -170366
b) -374239
c) 211442

ii) Only the decrement method is synchronized, these are some of the results:

a) -549233
b) 45585
c) -2860

iii) Only the increment method is synchronized, these are some of the results:

a) -48565
b) -27382
c) 3666052

iv) Both the increment and the decrement methods are synchronized, these are some of the results:

a) 0
b) 0
c) 0

From the results above, we can observe that, apart from the last case where both the increment and the decrement methods were synchronized, none of the other cases produced the expected result.

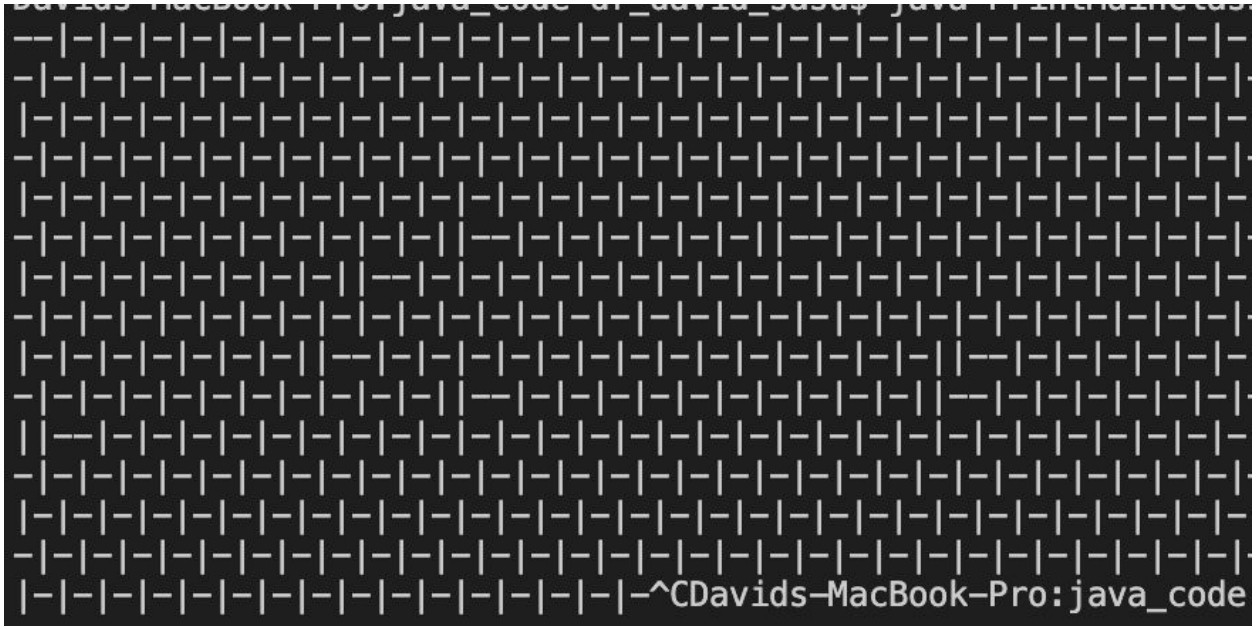
This is because, when both of the methods are not synchronized, the process of a running thread executing either of these two methods can be interrupted and this would lead to a final result which was not expected.

1.2

1. I have 8 cores on my mac. So this means that I can run 16 hardware threads in parallel.
2. If I change the range into a large number, it just takes a very long time for my program to finish its execution.

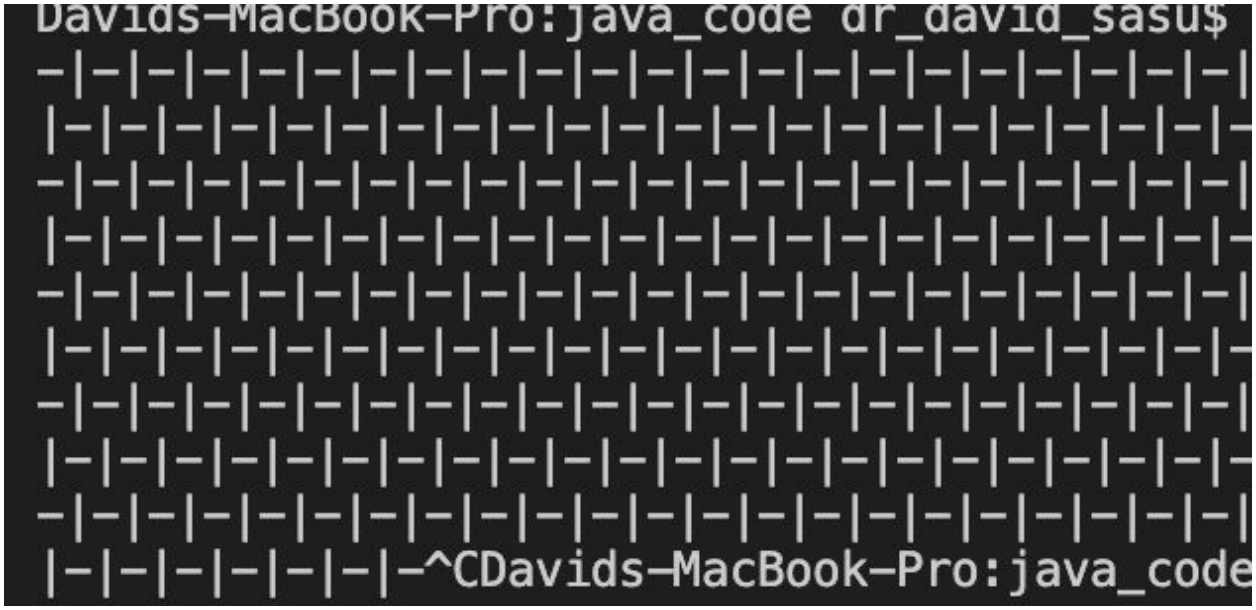
1.3

2.



The image above represents the output of my print program. From the image, we can observe that the bar and the dash symbols alternate, but occasionally, 2 dashes are printed next to each other or 2 bars are printed next to each other.

3.



From the image above, it can be observed that the printed bars and dashes now alternate perfectly without having scenarios where there are 2 bars or dashes printed next to each other.

This works because of the use of the ‘synchronized’ keyword on the print method. The ‘synchronized’ keyword prevents a thread from interrupting another thread when that thread is executing the method on the object instance.

4.

```
class Printer{
    private final Object myLock = new Object();

    public void print(){
        synchronized(myLock){
            System.out.print("-");
            try { Thread.sleep(50); } catch (InterruptedException exn) { }
            System.out.print("|");
        }
    }
}
```

5.

```
class PrintMainClass{

    public static void main(String args[]){

        Printer Print = new Printer();

        Thread t1 = new Thread(() -> {
            while(true){
                synchronized(Print){
                    Print.print();
                }
            }
        });
        Thread t2 = new Thread(()-> {
            while(true){
                synchronized(Print){
                    Print.print();
                }
            }
        });

        t1.start();
        t2.start();
    }
}
```

6. To ensure that the printing of the bars and the dashes alternate and there is not a situation where 2 bars are printed next to each other or 2 dashes are printed next to each other, the print function is synchronized to **bring order** to the activities of the threads that are executing that

method. This ensures that the activities of one thread **happens before** the activities of another. That is, one thread cannot interrupt another thread when that thread has not finished its process.

1.4

Green

The categories proposed by both Goetz and Kristen support each other and the both present the same ideas. Goetz's idea of 'resource utilization' supports Kristen's idea of 'exploitation of microprocessors'. This is because in both ideas, the idle resources of the computer are put to use to enable multiple computer programs to run together at the same time.
Also, Kristen's idea of 'Concealed Parallelism' supports Goetz's idea of 'Fairness'. This is because by ensuring that the different programs in the computer system are run in a parallel manner, there is fairness in the execution of the programs and one program does not get to hog all of the resources of the computer system.
Kristen's idea of 'Intrinsic Parallelism' supports Goetz's idea of 'Convenience'. This is because real world problems are often interconnected and complex, and as a result of this it is difficult to write one large program that can model all of the activities that are happening at the same time in parallel. Hence, it is usually convenient to write programs that are multi-threaded in nature.

Yellow

Exploitation of microprocessors:

- Windows Operating System
- Mac Operating System
- Linux Operating System

Concealed Parallelism:

- Windows Operating System
- Firefox Web Browser
- Web server Systems

Intrinsic Parallelism:

- Computer Games
- Text Editors
- Integrated Development Environments

Red

For the concept of the 'exploitation of microprocessors', when all of the microprocessors are used to execute different threads, concealed or intrinsic parallelism is able to occur.
Also, the concept of 'concealed parallelism' supports the concept of 'intrinsic parallelism' since we can write programs that can connect to multiple devices and share the same computing resources at the same time.

1.5

1.These are the results that I get when I run the program 4 times:

- Sum is 1842174.000000 and should be 2000000.000000
- Sum is 1835455.000000 and should be 2000000.000000
- Sum is 1835305.000000 and should be 2000000.000000
- Sum is 1865294.000000 and should be 2000000.000000

Since these results indicate that we do not get the final value that we desire when we execute the code, we can conclude that the class Mystery is **not** thread-safe.

2.Since the addInstance method and the addStatic method are both performing the same activity(i.e incrementing the value of sum by 1), even though thread t1 (which implements addInstance) may prevent other threads from executing the addInstance method on the instance of the class, when thread t1 is interrupted by thread t2 (which implements addStatic), thread t2 can still perform the same activity that thread t1 was performing because it has a lock on the entire class. Because of this reason, the ability of only one thread to increment sum at a time is not realized.

3. We could make the class thread safe by placing an explicit lock on the Object m. This would ensure that only one thread can call a function on the object m at a time. This is demonstrated below:

```
public class TestingLocking0 {
  public static void main(String[] args) {
    final int count = 10;
    Mystery m = new Mystery();

    Thread t1 = new Thread(() -> {
    for (int i=0; i<count; i++){
        synchronized(m){
            m.addInstance(1);
        }
    }

      });


    Thread t2 = new Thread(() -> {
    for (int i=0; i<count; i++){
        synchronized(m){
             m.addStatic(1);
        }
    }
      });


    t1.start();
    t2.start();
    try {
        t1.join();
        t2.join();
        } catch (InterruptedException exn) { }
    System.out.printf("Sum is %f and should be %f%n", m.sum(), 2.0 * count);
  }
}
```

1.6

1. Even though the lock is being applied to the MysteryB class, no lock is being applied to the MysteryA class. So as a result of this, if thread t2 should interrupt the execution of thread t1, it can still increment the value of count since the increment method belongs to the MysteryA class and not the MysteryB class.

2. An explicit lock object can be created in the MysteryA class and synchronized statements can be placed around the incrementation operations for the count variable in both the increment and the increment4 methods. This is demonstrated below:

```
class MysteryA {
  protected static long count = 0;
  public static final Object myLock = new Object();

  public static synchronized void increment() {
    synchronized(myLock){
        count++;
    }
  }

  public static synchronized long get() {
    return count;
  }
}

class MysteryB extends MysteryA {
  public static synchronized void increment4() {
    synchronized(myLock){
        count += 4;
    }
  }
}
```

1.7

The main difference between the two ways to use synchronized is that applying a synchronized statement on this ‘this’ keyword **involves** the use of a monitor while just making an instance method synchronized **does not involve** the use of a monitor.