

# Concurrency: Writing Multithreaded Programs in Java

Melvyn Ian Drag

November 20, 2019

## Abstract

Up until now we've written **serial** code. Today we will learn to write **concurrent** code. In other words, before we only programmed our machines to do one thing at a time. Now we will write programs that multitask - or at least give the illusion of multitasking.

## 1 Introduction

Concurrency seems like a simple concept - you can program a computer to do several things at once. If your computer is using a multicore processor like an intel-iX then it's very easy to image a computer doing simultaneous/concurrent operations, because there are acutally multiple CPU cores working at once. But even single cores can give the illusion of doing things concurrently! You will learn about how all our fancy modern operating systems like Windows and Linux and MacOS do this via "Context Switching".

Nevertheless, the business of actually *programming* your computer to do things concurrently is quite challenging and takes years of thought and disciplined practice to do correctly. Don't take my word for it, take *Enrique Lopez Manas*' (some google developer I found on medium.com) word for it:

*Multithreading is an entire discipline that takes years to master and properly understand. We will keep a short introduction in this article*

Link: <https://medium.com/google-developer-experts/on-properly-using-volatile-and-synchron>

As he said in the intro to the article linked above - we too will keep our intro short and sweet. The purpose of this lecture is to get your "toes in the water" so to speak, and then you can go on the internet and research more about concurrent programming in Java.

### 1.1 What we'll learn tonight

We are going to learn about:

1. Threads
2. Runnablees
3. synchronized statements
4. synchronized functions
5. Atomic variables
6. Take another look at primitives

## 2 Get the Class Repo

Make sure to download the latest iteration of the class repository so you can get all the code we are going to be using. You could download the files one by one throughout the night, but it is easier to just download everything.

## 3 The problem of the Night

### A reduce operation

A reduce operation is one where you take a big list of values and reduce them down to a single value. A common reduce operation is to sum all the numbers in a List. Another is to find the largest value in a List. Or the smallest. Or multiply all the numbers.

Tonight we will focus on the sum operation. You should already know how to do this serially - that is, with one thread.

Here is the code you might come up with:

```
import java.util.*;
public class SerialSum{
    public static void main(String[] args){
        final int N = Integer.parseInt(args[0]);
        int total = 0;
        List<Integer> ints = new ArrayList<Integer>();
        for( int i = 0; i < N; ++i ){
            ints.add(i);
        }
        final long T0 = System.nanoTime();
        for( Integer i : ints ){
            total += i;
        }
        final long TF = System.nanoTime();
        System.out.println(String.format(
            "The sum of the integers in [%d, %d) is %d.",
            0, N, total )
        );
        System.out.println(String.format(
            "The sum took %d nanoseconds to compute.",
            TF - T0)
        );
    }
}
```

And here are the results of running the code with various inputs. Compare your machine's performance to mine to see how it compares to a T450 with an i5 processor and 8GB ram.

And we can quickly verify that this silly little computation is correct by using a little trick from

```
The Core View Search Terminal Help
melvyn$ java SerialSum 5
The sum of the integers in [0, 5) is 10.
The sum took 872 nanoseconds to compute.
melvyn$ java SerialSum 5
The sum of the integers in [0, 5) is 10.
The sum took 832 nanoseconds to compute.
melvyn$ java SerialSum 50
The sum of the integers in [0, 50) is 1225.
The sum took 1478 nanoseconds to compute.
melvyn$ java SerialSum 50
The sum of the integers in [0, 50) is 1225.
The sum took 1367 nanoseconds to compute.
melvyn$ java SerialSum 5000
The sum of the integers in [0, 5000) is 12497500.
The sum took 83616 nanoseconds to compute.
melvyn$ java SerialSum 5000
The sum of the integers in [0, 5000) is 12497500.
The sum took 83705 nanoseconds to compute.
melvyn$
```

Figure 1: Testing summing numbers from 0 to 4, 49, 4999.

math class:

$$\sum_{i=0}^{N-1} i = \frac{(N-1) * N}{2} \quad (1)$$

And so, according to equation 1, we can compute the sums we just tested in Java.

$$\begin{aligned} \sum_{i=0}^4 i &= \frac{(4) * 5}{2} \\ &= 10 \\ \sum_{i=0}^{49} i &= \frac{(49) * 50}{2} \\ &= 1225 \\ \sum_{i=0}^{4999} i &= \frac{(4999) * 5000}{2} \\ &= 12497500 \end{aligned}$$

Look at figure 3 and compare the outputs there to the values we computed with our little math trick. We should all now agree that our simple SerialSum.java is working correctly - but can we make it *faster*? Of course we can, that's the purpose of tonight's lecture!

## 4 Threads

To do concurrent things, Java uses **Threads**. So far our code has been single-threaded ( aka 'serial' ). Multithreaded applications, like we will write now, are referred to synonymously as 'concurrent' programs.

## 4.1 Create Threads by Subclassing *Thread*

Note that the our simple class *SubclassThread* contains a *run()* method. Do not call that method. That is the function that is called when you call *.start()*.

### Attention!

Read the above and the code sample carefully. You DO NOT call the *run()* method yourself. That function is called when you call *.start()*.

```
public class SubclassThread extends Thread {
    private static int numThreads = 0;

    private int threadNum;

    SubclassThread(){
        threadNum = numThreads++;
    }

    public void run(){
        System.out.println(String.format("Hello from thread %d",
            this.threadNum));
    }

    public static void main(String[] args){
        Thread t0 = new SubclassThread();
        Thread t1 = new SubclassThread();
        Thread t2 = new SubclassThread();
        Thread t3 = new SubclassThread();
        Thread t4 = new SubclassThread();

        t0.start();
        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }
}
```

Note that the order in which the threads execute the *run()* method is non-deterministic. Each run, they execute in a different order. This is something to be aware of. For today's lecture it doesn't matter, but for other code you write you will have to be careful because of this. Refer to image 4.1 to see this code in action.

## 4.2 Create Runnableables by Subclassing *Runnable*

Whereas *Thread* is a class, *Runnable* is an interface. When you do the homework, you will read about the relative merits of using the class vs the interface. For now, we are just going to write a couple of simple examples and use them.

Note that the class that implements *Runnable* still has to define a *run()* method. Just as with the *Thread* class, when we call *.start()*, the *run()* method is called.

```
melvyn$ java SubclassThread
Hello from thread 2
Hello from thread 0
Hello from thread 3
Hello from thread 4
Hello from thread 1
melvyn$ java SubclassThread
Hello from thread 3
Hello from thread 4
Hello from thread 0
Hello from thread 1
Hello from thread 2
melvyn$ java SubclassThread
Hello from thread 4
Hello from thread 0
Hello from thread 3
Hello from thread 1
Hello from thread 2
melvyn$
```

Figure 2: Creating a class that extends ‘Thread’

```
public class SubclassRunnable implements Runnable {
    private static int numThreads = 0;

    private int threadNum;

    SubclassRunnable(){
        threadNum = numThreads++;
    }

    public void run(){
        System.out.println(String.format("Hello from thread %d",
            this.threadNum));
    }

    public static void main(String[] args){
        Thread t0 = new Thread(new SubclassRunnable());
        Thread t1 = new Thread(new SubclassRunnable());
        Thread t2 = new Thread(new SubclassRunnable());
        Thread t3 = new Thread(new SubclassRunnable());
        Thread t4 = new Thread(new SubclassRunnable());
        t0.start();
        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }
}
```

Note that the order in which the threads execute the *run()* method is non-deterministic. Each

```
melvyn$ java SubclassRunnable
Hello from thread 3
Hello from thread 4
Hello from thread 2
Hello from thread 0
Hello from thread 1
melvyn$ java SubclassRunnable
Hello from thread 1
Hello from thread 0
Hello from thread 3
Hello from thread 4
Hello from thread 2
melvyn$ java SubclassRunnable
Hello from thread 1
Hello from thread 0
Hello from thread 4
Hello from thread 2
Hello from thread 3
melvyn$
```

Figure 3: Creating a class that extends ‘Runnable’

run, they execute in a different order. This is something to be aware of. For today’s lecture it doesn’t matter, but for other code you write you will have to be careful because of this. Refer to image 4.2 to see this code in action.

### 4.3 You need to read more

I am confident discussing this topic because I’ve read a stack of references. To cement your knowledge of what I’ve just said about threads, and about what I am about to say, you 100% should read the official Java tutorial about threads.

It is here:

<https://docs.oracle.com/javase/tutorial/essential/concurrency/>

After today’s lecture, this material should be digestible.

## 5 The *synchronized* keyword

I’ve already told you - mastering concurrency takes time. You need to learn about many errors that could arise and learn how to mitigate them. There is no time to address them all, so I’ll present a basic one to you. This is a very intellectually rewarding ( and financially if you can master the topic and get a job doing it ) topic.

A major problem when writing concurrent code is ensuring that two competing thread don’t access computer memory at the same time. If you are in class and reading this, make sure to ask me to draw a picture on the board to illustrate the problem of competing memory accesses.

Draw a diagram showing that two threads accessing the same memory location without synchronization can cause mistakes in your code. This is one of the most common and insidious problems you come across

## 5.1 The example of the day in code

For the heck of it I chose to implement this example by implementing the Runnable interface. You can tinker with this code and make it use the Thread base class. That would be a fun project for you. In general, this code is the same as our SerialSum example, except it uses a few threads to do the work instead of just one.

```
import java.util.List;
import java.util.ArrayList;
import java.util.concurrent.*;

public class SynchronizedSumReduce implements Runnable{

    public static int total = 0;

    private List<Integer> _myInts = null;

    SynchronizedSumReduce( List<Integer> ints ){
        _myInts = ints;
    }

    private synchronized void updateTotal( int partialSum ){
        total += partialSum;
    }

    public void run(){
        int partialSum = 0;
        for(Integer i : _myInts ){
            partialSum += i;
        }
        updateTotal(partialSum);
    }

    public static void main(String[] args){
        List<Integer> intList = new ArrayList<Integer>();
        final int N = Integer.parseInt(args[0]);
        for(int i=0; i < N; ++i){
            intList.add(i);
        }
        List<Integer> intList1 = intList.subList(0, N/4); // upperBound
                    is Exclusive
        List<Integer> intList2 = intList.subList(N/4, N/2); //
                    lowerBound is inclusive
        List<Integer> intList3 = intList.subList(N/2, 3*N/4); //
                    lowerBound is inclusive
        List<Integer> intList4 = intList.subList(3*N/4, N); //
                    lowerBound is inclusive
        final long T0 = System.nanoTime();
```

```

Thread t1 = new Thread(new SynchronizedSumReduce(intList1));
Thread t2 = new Thread(new SynchronizedSumReduce(intList2));
Thread t3 = new Thread(new SynchronizedSumReduce(intList3));
Thread t4 = new Thread(new SynchronizedSumReduce(intList4));
t1.start();
t2.start();
t3.start();
t4.start();
try{
    t1.join();
    t2.join();
    t3.join();
    t4.join();
}

catch( Exception ex ){
    System.err.println(ex.toString());
}
final long TF = System.nanoTime();
System.out.println(String.format(
    "The sum of the integers in [%d, %d) is %d.",
    0, N, SynchronizedSumReduce.total )
);
System.out.println(String.format(
    "The sum took %d nanoseconds to compute.",
    TF - T0)
);
}
}

```

## 5.2 A Problem! And a Solution! Use the Right Datatype

Note that an issue with this is if we try to run the code with parameter *1000000*. The *int* datatype cannot hold the sum of 0 to 999999, and overflows. Try to run the code and you will see the solution.

Change the *partialSum* and *total* variables to be *longs* instead of *ints* and then they can hold larger values.

See *MyCode/LongSynchronized/SynchronizedSumReduce.java*.

Of course this isn't a 100% solution - the long data type can also be overflowed! Java has classes to handle data that is larger than the 8 bytes of a long. I believe the big number classes are called *BigInt* or *BigNum* or *BigInteger* or the like.

Exercise: Have class find the name of the classes for integers larger than 8 bytes in Java

We aren't going to use those today, but they are super simple classes to use, if you want to goof around with them just look through the docs you find online and write some toy examples.



### 5.3 Forgetting to synchronize my threads caused an error!

Look at this code and note that the *updateTotal* method is not synchronized. Then look at the output of the code shown in figure 5.3 and note that the output is wrong. The right output can be computed using the formulas given in section 3

The code is in *MyCode/RaceConditions/NonSynchronizedSumReduce.java*. This code is the same as the code in the *SynchronizedSumReduce* example, except the *update* method was changed from this:

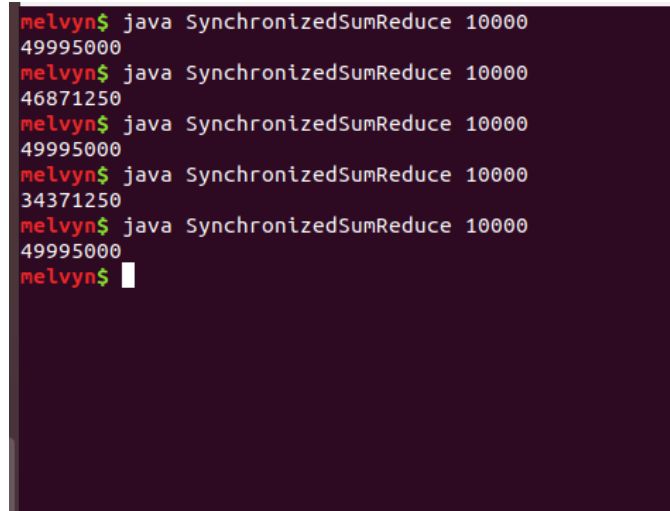
```
private synchronized void updateTotal( int partialSum ){
    total += partialSum;
}
```

to this:

```
private void updateTotal( int partialSum ){
    try{ Thread.sleep(10);}
    catch(Exception e){}
    total += partialSum;
}
```

Refer to figure 5.3 to see that the output now changes each run, because the threads are making memory accesses in a non synchronized manner.

Note to self - these notes need an image of race conditions. If you are reading them and are confused by race conditions, don't hesitate to contact me so I can explain it.



```
melvyn$ java SynchronizedSumReduce 10000
49995000
melvyn$ java SynchronizedSumReduce 10000
46871250
melvyn$ java SynchronizedSumReduce 10000
49995000
melvyn$ java SynchronizedSumReduce 10000
34371250
melvyn$ java SynchronizedSumReduce 10000
49995000
melvyn$
```

Figure 4: Creating a class that extends 'Runnable'

**Take away:** Whenever multiple threads are accessing some shared memory - in this case they are all accessing the shared ( static ) variable 'total' - make sure to synchronize their accesses.

## 6 Break and Experiment. 15 minutes.

Take some time now to experiment with what I have said. Note that I added a sleep to the raceCondition example. Try adding that sleep to the properly working example and notice that it doesn't affect the results. By doing this you will gain some good insight.

## 7 Another interesting thing - 10 minutes

Look at my race condition code. Note that I have broken up the sum computation into 4 worker threads. Each thread gets a slice of the total sum to compute. Here is the chunk of the ArrayList given to each thread:

```
List<Integer> intList1 = intList.subList(0, N/4); // upperBound is Exclusive
List<Integer> intList2 = intList.subList(N/4, N/2); // lowerBound is inclusive
List<Integer> intList3 = intList.subList(N/2, 3*N/4); // lowerBound is inclusive
List<Integer> intList4 = intList.subList(3*N/4, N); // lowerBound is inclusive
Thread t1 = new Thread(new SynchronizedSumReduce(intList1));
Thread t2 = new Thread(new SynchronizedSumReduce(intList2));
Thread t3 = new Thread(new SynchronizedSumReduce(intList3));
Thread t4 = new Thread(new SynchronizedSumReduce(intList4));
```

This didn't come from thin air - this is from the RaceCondition example. Note that in the image that shows the invalid sums computed by the nonsynchronized code. My question to you is - which threads did not execute properly? Which of the threads had his partialSum ignored in the computation of the total?

*After 10 minutes, do the math on the black board*

## 8 Timing

**Ask students: If a single thread can sum 100 numbers in 100ms, how long can TWO threads sum 100 numbers?**

*Answer should be half the time, two workers should cut the work in half, so 50ms*

In fact, with this simple example I've given you you will see that the timing is much much much worse with multiple threads running.

Run the code sample *MyCode/LongSynchronizedTiming* and note that the code runtime is much worse with multiple threads.

Have students verify the behavior on their computers.

This is strange and unexpected. This is because of **overhead**. The JVM is a complex thing and it seems with this example (on my computer at least), the JVM requires a lot of memory and CPU resources to perform this computation when using threads. You can look at my example and see if I've done something wrong! It could be an operating system bug.

I hoped to prepare this simple example for you to show you something basic you can do with multiple threads - and it didn't work! I've spent a few days tinkering with this example and I haven't been able to improve the run speed! If you have time you could tinker with the code, your JVM settings, maybe your operating system settings, and you could probably get the code to run faster.

I said the problem is **overhead** - and actually it also might **resource limitation** done by the JVM or OS. This interesting problem deserves more attention, but I am out of time to dedicate to it.

This is a very serious problem and I don't understand it.

## 9 The *volatile* keyword

Do not use this for your homework. I am just showing you this so that you know about it. *volatile* is a very complicated keyword that generates a lot of confusion in C, C++ and Java. The meaning of the keyword is slightly different between these languages, but roughly what it means is "don't allow any thread to make a local copy of this variable". This has the effect that the variable is synchronized when accessed by multiple threads. Have a look at the code in *MyCode/Volatile*. This is the same as the code we've been using, except `total` is now *volatile* and the *updateTotal* method is no longer synchronized. You should read more about this keyword as it is very complicated and error prone. I'm just showing it to you.

## 10 Atomic variables

Roughly speaking, atomic variables are variables that can only be modified by one thread at a time. This is similar to *volatile*, but also a bit different. I'll show you two atomic variable examples.

### 10.1 AtomicInt

I'll show you how to make this work with an `AtomicInteger`. `AtomicIntegers` are integers that protect integers so that they cannot be corrupted by multiple threads. The net effect is similar to the effect of marking the variable *volatile* or making functions synchronized. There are further implications, but, as always, you'll have to do some reading on your own if this interests you. There is certainly a reason why Java gives you multiple ways to protect global variables being modified by multiple threads. I'm not sure of the pitfalls of each of the methods, and even if I did I don't think it would do you any good if I told you. Since you're just learning concurrency now, it would just scare you. Remember -

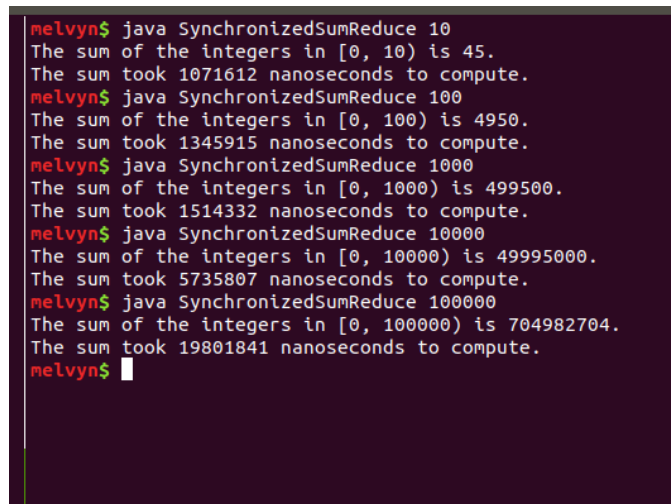
**Concurrency takes years to master. Don't take my word for it, I'm just an adjunct with a master's degree! I showed you the same words coming from a professional at google.**

**This is why it takes years! You have to learn all your options for concurrency, then learn of the dangers of each option, and then practice a lot.**

Look at the code in *MyCode/AtomicInt*. Note that it doesn't use the `Synchronized` keyword. Figure 10.1 shows the code running successfully and shows that the total overflows when we try to compute the sum from 0 to 99999. This is because, as we discussed before,

$$\begin{aligned}
\sum_{i=0}^{99999} i &= \frac{99999 * 100000}{2} \\
&= 99999 * 50000 \\
&= 4999950000
\end{aligned}$$

Since the sum is larger than the largest *int*, 2147483647, it overflows. You can remedy this by using a larger atomic type, *AtomicLong*.



```

melvyn$ java SynchronizedSumReduce 10
The sum of the integers in [0, 10) is 45.
The sum took 1071612 nanoseconds to compute.
melvyn$ java SynchronizedSumReduce 100
The sum of the integers in [0, 100) is 4950.
The sum took 1345915 nanoseconds to compute.
melvyn$ java SynchronizedSumReduce 1000
The sum of the integers in [0, 1000) is 499500.
The sum took 1514332 nanoseconds to compute.
melvyn$ java SynchronizedSumReduce 10000
The sum of the integers in [0, 10000) is 49995000.
The sum took 5735807 nanoseconds to compute.
melvyn$ java SynchronizedSumReduce 100000
The sum of the integers in [0, 100000) is 704982704.
The sum took 19801841 nanoseconds to compute.
melvyn$

```

Figure 5: Creating a class that extends ‘Runnable’

## 10.2 Exercise

Change the above code to use an *AtomicLong* instead of an *AtomicInteger*. Verify that you can sum from 0 to 99999.

The things you’ll need to change are:

1. Change all *AtomicIntegers* to *AtomicLongs*
2. Change *partialSum* to a *long*
3. Change the parameter to *updateTotal* to a *long*

## 10.3 Last Note about Atomic Types

There seems to have been some enhancements done around the release of Java 8. Look at the docs:

Methods added for Java 8:

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicLong.html>

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/AtomicInteger.html>

Compared to Java 7:

<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicLong.html>  
<https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicInteger.html>

## 10.4 If time to kill

The overflow when we tried to sum 0 to 99999 in an integer was 7xxxxxxx. Using knowledge of twos complement and binary addition, does that make sense?

$$4999950000 - 2147483648 = 2852466352 \quad 2852466352 - 2147483648 = 704982704$$