# CISC 3320 HW Assignment – 3 (7 pts)

This assignment accentuates the importance of synchronization when there is concurrency. You are asked to manufacture a situation where data inconsistency results from multiple threads trying to write into a common piece of data. Then you write another version of the same program that solves the issue present in the previous version (the difference is quite small)

## Requirements

1. Use either C/C++ or Java to write two programs: `SyncDemo1` and `SyncDemo2`
2. In `SyncDemo1`
   - Populate an array or `ArrayList` with integers 1~1000.
   - Calculate the sum of all its elements – you do this by creating 5 threads and passing to each thread 1/5 of the data (i.e., one adds 1~200, another adds 201~400, and so on…)
   - In HW-2 you are recommended to use threads that implement `Callable` so they can return their result. In this exercise, you need threads that implement `Runnable` and add their subtotal into a grand total, which is not protected with synchronization.
   - Your main thread then prints the grand total, and it should show an incorrect result (the correct sum of 1+2+…+1000 = 500,500)
3. In `SyncDemo2`
   - Fix the data inconsistency issue exhibited in `SyncDemo1` by employing synchronization measures, so that the sum reported is correct.

## Hints

With C/C++ the grand total would be a global variable. With Java, one can use a static variable of the main class. Such a variable can be accessed by a static inner class which will be your task class that implements `Runnable`.

To create data race, you may have to exaggerate the random sequencing of the updating process of the `grandTotal` variable. So instead of writing `grandTotal += subtotal`, you can use:

```
int temp = grandTotal + subTotal;
// wait a few milliseconds
grandTotal = temp;
```

With Java, you wouldn't be able to solve the problem just by adding a `synchronized` keyword in front of your thread's worker method definition, if each thread instance is independent. That's because this feature makes use of the thread object's intrinsic lock, and 5 threads will have 5 independent intrinsic locks – losing protection for any individual thread. There are several possible solutions. You can explicitly use a single `ReentrantLock`. You can also give each thread the access to a common object, and make the `synchronized` function a member of that shared object so they all use the same intrinsic lock (this is the case with the `BoundedBuffer` example in the slides). Alternatively, note that the synchronized keyword can be used to create a *synchronized block*:

```
synchronized (expr) {  // expr must evaluate to an object reference
    statements;
}
```

In this case, the statements inside the block will acquire the lock on the object represented by `expr` (which could be any shared object), not the individual thread instance. In all, using `ReentrantLock` is probably the easiest and most straightforward approach.

## What to turn in

- Please submit (via email) the following:
    - your source code in separate files
    - running result (you could use a screen capture)
- Please do not copy code, whether from classmates or from the Internet.