# Timing Report

## Sample code unzipped

The run times for the sample code are below, given in Nano-seconds.

#1 - 2519065800
#2 - 2512737000
#3 - 2509407699
#4 - 2519603101
#5 - 2506720401
Average speed - 2513506800.2
Slowest Speed - 2519603101
Fastest speed - 2506720401

## Synchronized code

The sample code was refactored to utilise the synchronized keyword. This fixes the bug from the original code. The run times are shown below, given in Nano-seconds

#1 - 2507644700
#2 - 2507580800
#3 – 2506843400
#4 - 2526980500
#5 - 2513147800
Average speed - 2512439440
Slowest speed - 2526980500
Fasted speed - 2506843400

## Mutex lock code

The sample code was refactored to use a ReentrantLock object to act as a mutex lock in the critical section of the code. The run times are shown below, given in Nano-seconds.

#1 - 2521521701
#2 - 2509274699
#3 - 2511955700
#4 - 2518795199
#5 - 2519310100
Average speed - 2516171479.8
Slowest speed - 2521521701
Fastest speed - 2509274699

## Atomic Integer code

The sample code was refactor to utilize the AtomicInteger object to handle thread safety.
The run times are shown below, given in Nano-seconds.

#1 - 2507402600
#2 - 2511677199
#3 - 2508716900
#4 - 2508812101
#5 - 2507083700
Average speed - 2508738500
Slowest speed - 2511677199
Fastest speed - 2507083700

## Conclusion

By average speed, the fasted method of achieving thread safety is by using Atomic Integer object as a counter. The slowest being the Mutual Exclusion lock. However, with the extra over head of implementing Mutual Exclusion, the simple ‘synchronized’ keyword is nearly just as fast with less refactoring needed.