Calvin Liu
804182525

Environment:
Compiled java programs on UCLA linux server using java 8. SSH'ed via
MacBook Pro (13-inch, Mid 2012)
2.5 GHz Intel Core i5
4 GB 1600 MHz DDR3

Background:
The purpose of the experiment is to time the speed in which a number of 'swaps' provided by a function is done on an array of numbers. The process was timed until it was completed and
The correlation of the time/transition seems to be:

- The less number of swaps, the more time per transition
- The less number of threads, the less time per transition

when using the test harness with different values such as changing the number of threads, the amount of swaps, the number of elements in the array.

```
Decreased threads:
java UnsafeMemory GetNSet 8 1000000 100 10 20 30 40 50 60 70 80 90 100
Threads average 1601.33 ns/transition
Decreased Swaps:
java UnsafeMemory GetNSet 16 10000 100 10 20 30 40 50 60 70 80 90 100
Threads average 26275.5 ns/transition
```

Testing:
Parameters: 16 threads, 1,000,000 swaps, array = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100], maxval 100. If a maxval is not given then the default value is 127.
Tests were run 10 times and averaged.

Synchronized: ~5536.1 ns/transition, 100% reliable. DRF because threads can't run swap simultaneously. This is why the time per transition is higher than some of the other states.

```
java UnsafeMemory Unsynchronized 16 1000000 100 10 20 30 40 50 60 70 80
90 100
```

Null: ~2039.9 ns/transition, 100% reliable. DRF but just because it doesn't really do anything.

```
java UnsafeMemory Null 16 1000000 100 10 20 30 40 50 60 70 80 90 100
```

Unsynchronized: ~2436.3 ns/transition Not at all reliable. Non-DRF because there is nothing to check for race conditions

```
java UnsafeMemory Unsynchronized 16 1000000 100 10 20 30 40 50 60 70 80
90 100
```

GetNSet: The original test case of 16 threads with 1000000 swaps was too much for the function to handle and it seems to hang a lot. In order to get a more accurate turnout of the test I changed the threads to 8 and the number of swaps to 10. This showed that GetNSet did succeed and did fail while being in between unsynchronized and synchronized. The average time of its run is 8.015404e+06 compared to (with same parameters) Unsynchronized: 7.416781e+06, Synchronized: 8.235532+e06. This shows that it is in between synchronized and unsynchronized. This model is Non-DRF because it must get values and then set values which leaves room for race conditions sometimes.

```
java UnsafeMemory GetNSet 8 10 100 10 20 30 40 50 60 70 80 90 100
```

BetterSafe: ~3094.9 ns/transition, 100% reliable. DRF because I used the ReentrantLock function in order to prevent race conditions. It is faster than synchronized because a ReentantLock has the ability to interrupt threads while waiting for a lock. The synchronized version can be blocked waiting for a lock for a long period of time.

```
java UnsafeMemory BetterSafe 16 1000000 100 10 20 30 40 50 60 70 80 90
100
```

BetterSorry: The BetterSorry implementation is faster than the BetterSafe because The BetterSafe method uses ReentrantLocks which take longer in making sure the implementation is 100% reliable, free from race conditions. I also used getAndIncrement and getAndDecrement in order to make sure each step was atomic and fast. BetterSorry is non-DRF and a race condition that occurs is that if you give it a large amount of swaps, a lower number of integers in the array, higher amount of threads, there is a higher chance that there will be a sum mismatch:

```
java UnsafeMemory BetterSorry 16 1000000 100 10 20 30 40 50 60 70 80 90
100
Threads average 2240.40 ns/transition
sum mismatch (550 != 763)
```

While if I just did 100 swaps, I get consistent reliability:

```
java UnsafeMemory BetterSorry 16 100 100 10 20 30 40 50 60 70 80 90 100
Threads average 1.79046e+06 ns/transition
An example program to run is:
java UnsafeMemory BetterSorry 24 1000000 100 70 80 90 100
```

Conclusion:

Null > Synchronized > BetterSafe > BetterSorry > GetNSet > Unsynchronized for reliability.

For the GDI application, I think it is best to use BetterSafe or BetterSorry depending on how well you want the output vs performance of the application to be. If you want the application to be perfect than BetterSafe would be best as it is faster than Synchronized and is 100% reliable. Otherwise, you can use BetterSorry whose performance is better than BetterSafe, but the reliability is not 100%. The more realistic state that would be used is BetterSorry.