My BetterSafe implementation used a ReentrantLock to provide thread-safe code. I used the lock from `java.util.concurrent.locks.ReentrantLock` and locked within the swap function only when I was incrementing and decrementing the byte array's values. This implementation was better than Synchronized's since I had finer grained locking. Instead of synchronizing the whole method with the keyword synchronized, I chose to lock only the parts where updates to the array were made. So the time at which there was locking is shorter and only done where needed. Doing it this way makes it reliable because the ReentrantLock will prevent more than one thread from accessing the locked parts of the code. Since we only care about the part were we increment and decrement the array elements in swap, that should be the part we lock and keep thread-safe.

To compare the performance, I tested on 30 threads, 1000 transitions, a maxval of 100, and an array of [10, 5, 2, 2, 1, 5, 2, 5, 63, 1]. The average runtime over 15 samples is:

	Performance (ns/transition)
Synchronized	9.10190E+06
BetterSafe	8.40895E+06

I also tested on the same parameters but instead with 1,000,000 transitions, getting average performances of:

	Performance (ns/transition)
Synchronized	8105.02
BetterSafe	4842.34

Thus BetterSafe has better performance while also retaining reliability.

My BetterSorry implementation used an array of AtomicInteger objects from `java.util.concurrent.atomic.AtomicInteger`. This provided even finer grained locking mechanisms since each element in the array could be independently locked and thus this incrementing and decrementing could be paralyzed instead of forcing the 2 lines of code to run sequentially as it is in BetterSafe and Synchronized. I used the AtomicInteger functions `getAndIncrement` and `getAndDecrement` which atomically added or subtracted 1 from the array element. The performance gain is noticeable for smaller number of transitions. I ran the program with 30 threads, 1000 transitions, a maxval of 100, and an array of [10, 5, 2, 2, 1, 5, 2, 5, 63, 1]. I got the following performance:

	Performance (ns/transition)
BetterSorry	7.68678E+06
BetterSafe	8.40895E+06

But for a larger number of transitions, like when I ran it with 1,000,000 transitions, BetterSorry's performance seemed to be worse than BetterSafes:

	Performance (ns/transition)
BetterSorry	6878.40
BetterSafe	4898.54

This pattern may be due to contention over more frequent locking.

My BetterSorry implementation is more reliable than Unsynchronized since I ran into a few race conditions, but not nearly as many compared to the unsynchronized version. BetterSorry still has mechanisms to prevent race conditions, so it is more reliable. I ran into race conditions with the following commands:

```
$ java UnsafeMemory BetterSorry 30
1000 100 10 5 2 2 1 5 2 5 63 1
Threads average 198454 ns/transition
negative output (-1 != 0)

$ java UnsafeMemory BetterSorry 2 10
5 5 4
Threads average 525500 ns/transition
output too large (6 != 5)
```

Race conditions seemed to occur the most when maxval is small, but overall it was quite difficult to observe any race conditions in BetterSorry. In comparison, Unsynchronized ran into race conditions quite often when some input array was given by the user.

Analysis of Various Classes:

Unsynchronized was definitely not DRF. There were no synchronization mechanisms to prevent race conditions and problems arise quite often when an array was input by the user. An example of a race condition I observed was:

```
$ java UnsafeMemory Unsynchronized
40 100 50 1 4 6 7 10
Threads average 3.26564e+06 ns/
transition
sum mismatch (28 != 30)
```

GetNSet was DRF. I used an AtomicIntegerArray where the whole array was locked during any writes to it. So it could not have experiences race conditions.

BetterSafe was DRF. I used a ReentrantLock in the swap function which surrounded the increment and decrementing operations to the byte array. This would protect the array from having multiple threads read the value and inaccurately update it. So I did not run into any race conditions.

BetterSorry was not DRF. Because I used an array of AtomicIntegers, only 1 thread is able to access an individual element at a time. But because the whole function is not thread-safe, the values in the array could change between where values are read and when they are incremented and decremented. An example of where a race condition occurred was:

```
$ java UnsafeMemory BetterSorry 2 10
5 5 4
Threads average 525500 ns/transition
output too large (6 != 5)
```

Overall it was difficult to create race conditions with BetterSorry, so running it with these parameters does not guarantee race conditions to occur.

To measure each implementation, I had to overcome the high standard deviation in my measurements. I believe this was due to variants in the load of the SEASNet servers and also in the threads created in each execution.

As for which class would be best for GDI's purposes, I think GetNSet would be best since it has the overall best performance.

Summary of Performances:

*30 threads, 1000 transitions, 100 maxval, array: [10 5 2 2 1 5 2 5 63 1]

Class	Performance (ns/transition)
Null	213719
Synchronized	9.10190E+06
Unsynchronized	226432
GetNSet	245388
BetterSafe	8.40895E+06
BetterSorry	7.68678E+06

*30 threads, 1,000,000 transitions, 100 maxval, array: [10 5 2 2 1 5 2 5 63 1]

Class	Performance (ns/transition)
Null	7661.17
Synchronized	7317.54
Unsynchronized	7046.83
GetNSet	7565.30
BetterSafe	4898.54
BetterSorry	6747.99