The sample test case is insufficient to accurately test the models. Because there is an inherent overhead of performing the test, I chose a higher number of operations (10 million swaps) to lessen the contribution of the overhead to the time per transition metric. I also chose a higher value for the number of threads (20), max value (100), and number of random array elements (600). The higher number of threads will emphasize the effects of multithreading. Given a fixed array size, the chance for collision will increase with the number of threads because more threads are now trying to access the same number of elements. Therefore, I upped the size of the array to avoid/minimize collisions. I chose a higher max value to allow for a greater range of values.

Synchronized
The given Synchronized is reliable/DRF because of the synchronized swap method that guarantees only a single thread can execute the block of code. Performance-wise it is the slowest of all the models because of the same reason why it is reliable.

Unsynchronized
As expected, the Unsynchronized model is unreliable due to the absence of synchronization and atomic operations. Any threads can simultaneously access and change the element values in the array, which leads to race conditions. I saw a mismatch error percentage of about 15%. In addition, the data race can often lead to deadlocks, especially if the initial values in the array are set to the edge values, 0's and maxval's. This makes it more probable that after certain number of swaps, all the values in the array are 0 or maxval, causing the swap to always return false and a deadlock.

GetNSet
I implemented the GetNSet model using AtomicIntegerArray and the Get and Set methods mentioned in the spec. Because the AtomicIntegerArray class accepts integer array as input, I converted the byte array into an integer in the constructor and vice versa when returning the array. GetNSet is not reliable because of the separate atomic Get and Set operations. Threads could be interweaved in a way that two or more threads read, decrement/increment, and set the same memory location concurrently, leading to a data race. A slightly better alternative would be to use the getAndDecrement/Increment methods so the two separate atomic operations are combined into one. The spec says that GetNSet should have performance that is halfway between Unsynchronized and Synchronized model but mine is definitely close to Unsynchronized. The sum mismatch error percentage is slightly lower than unsynchronized, at less than 10%.

BetterSafe
I tried many things for BetterSafe, all of which should yield a performance gain but none actually did. I used an array of locks, one for each element. Intuitively, I thought it would improve performance but the results were a few times slower than the Synchronized model. I suppose that the overhead of that many locks is too great to overcome the concurrency benefits. I ended up using a very simple method. I used a single lock to lock the block of code with the swap method. Theoretically, this achieves the same thing as the synchronized keyword but I saw a performance speedup of 2x. This was surprising to me. I

guess that because the block of code/number of operations is small, it is lighter weight to use a lock than synchronized keyword.

BetterSorry

For BetterSorry, I used an array of AtomicInteger to better exploit concurrency because the AtomicIntegerArray is a thread-safe array. Only one thread has exclusive access to the array at a time. By changing to an array of AtomicInteger, the scope of atomicity shifted from the array level to the individual element level. Now concurrent threads can simultaneously access the array as long as the elements are different, thus improving performance. However, the downside is that it's not 100% reliable. The value of the elements could change between the checking if they're in range and decrementing/incrementing it. This can lead to race conditions in which threads are reading certain elements while others are setting the same ones, however the sum mismatch error rate is relatively low, less than 10%, when it does fail. It doesn't fail often when I use my normal test case but it does fail a lot more often with an array of all 0's and maxval elements. This model achieves better performance than BetterSafe and better reliability than Synchronized.

Results

Testing was on SEASnet Linux server with Java version 1.8.0_05. I ran each model 10 times with the same parameters and the time per transaction is summarized in the table below. As mentioned previously, the Null model has a relatively low time per transaction.

	Null	Synchronized	Unsynchronized	GetNSet	BetterSafe	BetterSorry
Mean	296.337	3335.747	1125.7172	1540.8537	1943.9724	1314.0312
Stddev	59.285	216.318	152.245	271.829	286.777	115.459

The results are all relative in the sense that it depends heavily on the test case and also the SEASnet environment. Even running the same test cases in the SEASnet environment at different time gives varying results. Therefore, these results are not absolute and only give a general sense of performance in relationship to one another.