This paper will discuss the pros and cons of different implementations of thread-safe classes in Java. Note that the class being implemented is a very simple shared array modification class. Specifically, the main method that will be of discussion is a

The first implementation we will discuss will use the java.util.concurrent package. Specifically, the implementation of BetterSafe using the concurrent package would have been based around the Semaphore class. By forcing each thread to acquire a semaphore before accessing and writing to the array, the accesses would have been atomic and thus data-race-free.

The advantages to using the semaphores would have been the ease of writing the problem. The code modification would have been minimal, just requiring the semaphore code to be added at the beginning and end of the swap method.

The disadvantage of using a semaphore is that semaphores can allow multiple threads to access the shared array. Since the shared array variable should only have one thread accessing at a time, the semaphore is essentially reduced to a lock, which is a special type of semaphore that only allows one thread to access the shared variable. Thus, the overhead of semaphores that allow multiple threads to access the shared variable is wasted, therefore making semaphores slower than locks.

The next implementation uses the java.util.concurrent.atomic package. In this implementation, the shared array will be implemented by an array of AtomicIntegers. By ensuring that each access to an element is atomic, we can ensure that the correctness and reliability of the multithreaded code is maintained.

The advantages to the java.util.concurrent.atomic package is that since the AtomicInterger class and many others in this package are public libraries, the source code is much more likely to be refined and optimal for multithreading and latency. The user does not have to worry about the correct implementation of concurrency and atomic code since the library takes care of that under the hood.

The disadvantage of the atomic package is the fact that the State interface works with byte arrays instead of integer arrays. The translation from byte arrays to AtomicIntegerArrays most likely slowed down the methods enough to make this implementation not worth the time. Since the swap method was so bare-bones in the implementation, the delays caused by other factors were exacerbated, causing the overhead of these conversions to matter.

The next implementation of BetterSafe uses the java.util.concurrent.locks package. Specifically, the ReentrantLock class was used in the package to ensure that only one thread accessed the shared array at a time. The implementation of a data-race-free swap method with ReentrantLock was simple, just involving the acquisition of the lock at the beginning of the method and releasing the lock right before the return statement.

The advantages of using ReentrantLock is similar to the advantages of semaphores. The implementation is quite simple and the overhead of using ReentrantLock is smaller compared to Semaphores and the synchronized block.

The disadvantage to using locks is that the speed increase of the method is not as large as other implementations. In effect, the reality is still that only one thread can work on the shared array variable at a time. Given the implementation of the swap method, each swap method only requires to atomically access two elements out of the total array. Therefore, if there were separate locks for each element of the array, the program could run in parallel more often, increasing the speed of the program. However, the overhead of creating and maintaining enough locks for each element in the array is very costly. In fact, the method is slower if each element of the shared array is governed by one lock due to the overhead. Another issue is the deadlock that can potentially occur when each thread secures two locks each; one lock for the number to subtract from and one lock for the number to add to.

Therefore, the only real way to implement a faster BetterSafe with locks is to just use one lock for the race-prone area of the method.

The last implementation of BetterSafe uses java.lang.invoke.VarHandle to modify the access modes of accesses to the shared variable. By creating a VarHandle for the shared array, we can modify the access modes to force threads to access the shared array in a volatile manner.

The advantage of this implementation is the speed of the eventual implementation. Since the variable handle only modifies the access modes of the shared variable, the code remains flexible to multithreading.

The disadvantage of the implementation is the complexity of the implementation. The VarHandle library is complicated and even after reading through the manual and the help page, I was left confused. I was left with a very vague understanding of how to use the class properly, so I decided not to use the package due to time concerns.

Out of the four implementations mentioned above, I decided to implement my solution using the ReentrantLock class in the java.util.concurrent.lock package. My implementation remained data-race-free due to the fact that the racy portion of the method is completely encased in a lock() and unlock() method calls, making the code section atomic. The implementation is faster than the synchronized block due to the lower overhead of the locks.

Threads	Sync	Unsync	GNS	BS
1	55	42	58	73
2	334	**160**	276	418
4	1088.4	**411**	641	591
8	2145	**1205**	2067	1107
16	3512	**5802**	4540	2289

Table 1: All times are in us. Bold entries indicate incorrect answer. All test cases ran 1,000,000 iterations and results are averages of 10 runs.

The results show that the Synchronized, GetNSet, and BetterSafe implementations are all reliable in a multithreaded environment. The Unsynchronized implementation is the only unreliable implementation, failing under all multithreaded environments. In terms of speed, all implementations depend on thread count with the GetNSet implementation being the fastest with low thread counts and BetterSafe being faster at higher thread counts.

From the results of the tests, it's easy to see the that the BetterSafe implementation performs better than all of the other implementations at higher thread counts. It is interesting to note that the BetterSafe implementation performs poorly at lower thread counts compared to the other data-race-free implementations. This is most likely due to the fact that in single threaded applications, the overhead of initialization becomes the bottleneck of performance. Since the BetterSafe implementation requires an initialization of a ReentrantLock unlike the other implementations, the BetterSafe implementation performs poorly in singly threaded runs.

For the purposes of GDI, I believe that the implementation of BetterSafe will be the best fit for the job. As seen by the results, the implementation scales better with more threads which is desired in an application with large amounts of data. The unsynchronized implementation fails far too often under a multithreaded environment, making it unsuitable for the job.

In brainstorming a solution to BetterSafe, I went through multiple iterations of data-race-free swap method implementations. The very first implementation involved having one lock for each array element. Although the implementation was data-race-free, the overhead of having so many locks and having to deal with deadlocks slowed down the solution to the point where the synchronized solution was faster. Even in the cases of having more array

elements, the synchronized solution was faster up til a decent number of array size.