Ariadna Shamraeva
CS 250
HW 2 Reports on tasks 1 -3

TASK 1:

This report examines the effect of using ‘volatile’ keyword vs. not using given keyword on the performance of a Java program. In Java, ‘volatile’ is used to state that a value of a given variable will be modified by different threads. It ensures that this value of a volatile variable is read from the main memory and not from the thread's local cache at all times. The goal of this experiment is to compare the performance of using keyword ‘volatile’ and non-volatile variables in a loop and determine whether it is proffered to use one way over another.

Methodology

For this experiment, our Java program Memory.java is executed with different sizes and experiment counts. Here, loop variable is the main variable of interest. It is tested as ‘volatile’ and non-volatile. The code written for this program calculates a running total by adding or subtracting the loop variable (determined by its parity). The operation was repeated a specified number of times (a.k.a. experiments) in order to gauge an average execution time for each scenario. Here, the smaller number in Average Execution Time (ms) is a more favorable outcome.

Results

For the sake of this experiment, we ran multiple variations of input of loop size to determine the difference. The table below represents the experiment results where bold indicates a smaller, more favorable, result:

Loop Variable Type	**Experiment Count**	**Size**	**Average Execution Time (ms)**	**Seed**
Volatile	200	250000	2.09376E-4	42
Non-Volatile	200	250000	**1.32378E-4**	42
Volatile	200	259000	2.16033E-4	42
Non-Volatile	200	259000	**1.3788E-4**	42
Volatile	200	99000	9.2261E-5	42
Non-Volatile	200	99000	**6.5079E-5**	42

Discussion:
The data demonstrates a noticeable difference in the execution times between the two ways of execution: ‘volatile’ and non-volatile. The data favors the non-volatile variables over the ‘volatile’ as the execution time is significantly smaller in the former (see table). It is as such due to added overhead of accessing main memory at each iteration in ‘volatile’. The greater the loop size, the more pronounced the performance gap becomes. It demonstrates the impact of ‘volatile’ on computational efficiency in scenarios requiring frequent memory access.

Conclusion:
This experiment demonstrates and validates the hypothesis that using ‘volatile’ may cause additional overhead by forcing every read and write to bypass the cache and go directly to main memory. While ‘volatile’ is an important element for ensuring the consistency of data in multithreaded applications, we cannot overlook its impact on program’s performance. Using ‘volatile’ requires the understanding the trade-offs between data consistency and performance, therefore developers must proceed with caution.

TASK 2

Introduction
For the task two, we aim to analyze the complexity of time associated with accessing elements at different locations within an array in a Java program. To achieve this, we conducted a series of experiments to measure the access times for elements in the first 10% of an array and for randomly chosen elements of the last 10% it. Here, we used a random integer to initialize objects, then we repeated the process across several experiments to calculate average access times.

Methodology
We created an array of integer objects with a size defined by the user via a command line input. Using a specified seed for the random number generator, the array was populated with random integers. This was done so that we assure consistency across the runs of the experiment. The seed is also entered by user using command line. For each experiment, we measured the following:
1. The time to access each element in the first 10% of the array.
2. The time to access a single, randomly selected element from the last 10% of the array.
3. Calculate sum of the elements.
We repeat these steps across all the experiments defined by user input. Here, average times were calculated to achieve a more accurate representation of data associated with the access times.

Results
The table demonstrates extremely low time to access the first 10% of the array (0-1 ns) while accessing the random 10% of it takes multiple hundreds of nanoseconds.

The results section would include tables and graphs. For example:

Experiments Count	**Size**	**Seed**	**Avg. time to access first 10%**	**Avg. time to access random last 10%**	**Sum**
1	25000000	42	1	416	-623797425765
200	25000000	42	0	972	-124659352565772
20	25000000	42	0	885	-12458814169800

Analysis
While analyzing the data, it was observed that the position of the array elements influences the access times for given array elements.
When it comes to accessing the first 10% of the array, accessing its elements sequentially typically benefits from cache locality. The access time here is reduced because sequential memory locations are likely to be loaded into the cache together.
Accessing a random element in the last 10% of the array, on the other hand, does not benefit from this cache locality. The random access pattern leads to cache misses which results in longer average access times.
In this experiment, the difference in access times illustrates the impact of cache efficiency and memory locality on performance.

Conclusion
This experiment highlights the significance of data locality in optimization of performance of Java programs. The first part of the experiment, demonstrates that sequential access patterns, as seen in the first 10% of the array, are significantly faster due to cache locality. Yet the random access patterns can lead to increased latency. It is beneficial to understand these characteristics as this knowledge can be helpful in writing more efficient Java code, especially in scenarios where performance is critical.

TASK 3

Introduction
For the third task, we explored the performance characteristics of two different Java collections: TreeSet and LinkedList. The comparison is done by measuring the time taken to find elements within these data structures. The goal of this comparison is to understand how each collection's underlying data structure influences its performance in terms of element access times.

Methodology
We populated both, a TreeSet and a LinkedList, with a range of integers 0 to 'sum'-1. The experiments that we conducted using these collections is such that each time we randomly selecting a number within this range and measuring how long it took to check if the number was contained in each collection. Here, we used the 'contains' method to achieve this this task, then the operation was repeated for a specified number of experiments to calculate average access times.

Results
In the table we may see that TreeSet demonstrates relatively fast access times while LinkedList's access times are significantly higher and keep growing as the list grows larger. TreeSet, on the other hand, maintains faster times regardless of the growth of the set size.

Experiments Count	Size	Seed	TreeSet	LinkedList
1	25000000	42	35291	36822041
200	25000000	42	6840	31933826
20	25000000	42	10864	40743960

Discussion
The *TreeSet* in Java is implemented as a Red-Black tree, a balanced binary search tree. This structure offers O(log n) time complexity for common operations like add, remove, and contains. The average time to find an element in the TreeSet reflects this logarithmic complexity, showing relatively quick access times even as the size of the set grows.
The *LinkedList* in Java is a doubly linked list. Operations that require searching through the list, such as contains, have a time complexity of O(n). The average time to find an element in the LinkedList is significantly higher than in the TreeSet, especially as the list grows larger, reflecting its linear search time complexity.
The experiment demonstrates that TreeSet is highly efficient for search operations compared to LinkedList. However, LinkedList offers its own set of advantages, such as constant-time insertions and deletions, which were not the focus of this task.

Conclusion
The choice between TreeSet and LinkedList must be dictated by the specific needs of the program in question. In cases where quick search, insertion, and deletion are crucial, and the data is constantly changing, a TreeSet may be more an appropriate option. On the other hand, if the application requires frequent insertions and deletions at the list's ends, and search operations are less critical, a LinkedList might be the better choice.
In the case with these Java collections, understanding the underlying data structures and their complexities is key to making informed decisions that optimize performance based on the application's specific requirements.