ASSIGNMENT 1

Myra Tariq (300301040)

University of Ottawa

SEG 2105[A]

October 11, 2023

Exercise 2

Question 1

	Memory Efficiency	**Computational Efficiency**	**Simplicity**	**Maintainability**
Design 1	Con: requires typeCoord, still more efficient then design 4, but less efficient than the rest	Con: inefficient as it needs to compute conversion depending on type input, but makes same number of conversion as design 2 and 3 (or less)	Pro: Stores both cartesian and polar points when needed, straightforward design, only minorly complicated aspect is the conversion calculations which are needed	Con: due to the type coordinate system it would be difficult to modify the program without affecting other aspects Con: due to the number of calculations/ type changes required for different methods its more likely that a programmer would make mistakes w/ design 1 compared to other designs
Design 2	Pro: Uses less memory than design 1, 4 and 5 to store instances	Pro: minimal calculations needed (other than design 4) becomes inconvenient to retrieve cartesian coordinate	Pro: Clear separation of coordinate types provides simplicity, but dealing with cartesian points is inconvenient/difficult	Pro: simple to maintain as you're only dealing with one type of point (polar), rather than two.
Design 3	Pro: Uses less memory than design 1, 4 and 5 to store instances	Pro: minimal calculations needed (other than design 4), becomes inconvenient to retrieve polar coordinates	Pro: Clear separation of coordinate types provides simplicity, but dealing with polar points is inconvenient/difficult	Pro: simple to maintain as you're only dealing with one type of point (cartesian), rather than two.
Design 4	Con: Utilizes the most amount of memory for object's instances	Pro: no calculations need to perform conversions since it stores all values. Design 4 is technically the most efficient in this aspect	Pro: The simplest design, as it stores both points at all time (only needs basic get methods, i.e this.x), however it's impractical as it only stores values, meaning that the user must do conversions themselves	Con: Combined coordinate system makes it inconvenient to make significant changes May increase likelihood of mistakes while making modifications as there is no division of coordinate types
Design 5	Con: To change coordinate type, a new instance must be created (either design 2 or 3), not as efficient	Con: similar to design 1, requires computation depending on what type of coordinates is being used	Pro: strong adherence to OO design provides good structure, and improves simplicity in comparison to other designs	Pro: The most maintainable, as it adheres to OO principals the strongest (due to the abstraction), meaning you can change aspects of the polar coordinates without affecting the cartesian point system due to the subclasses

Question 2

Table 1 – Number of points created and called with respect to time for both designs

Time (seconds)	Number of points created and called	
	Design 1	Design 5
5	15,937,671	36,363,000
10	33,766,189	74,983,208
15	50,287,580	113,054,725
20	66,783,105	154,156,774
25	83,153,028	190,808,763

Table 2 - Statistical analysis for the number of points created during 10 seconds of execution

	Median (points)	Maximum (points)	Minimum (points)
Design 1	386,095,816	389,655,660	372,189,647
Design 5	383,901,391	387,227,951	377,014,443

Table 3 – Time necessary to create and call m number of points

Number of Points	Time Elapsed in milliseconds	
	Design 1	Design 5
100	50	4
1000	22	8
10,000	45	9
100,000	58	19
1,000,000	321	119
10,000,000	2852	1140

Table 4 – Statistical analysis for the time (nanoseconds) required to create of 10,000,000 points

	Median (points)	Maximum (points)	Minimum (points)
Design 1	0	9115500	0
Design 5	42	9086459	0

* I had to increase the number of points as the time was too short to produce only 100000 was too short (almost all values were 0 ms), additionally, had to switch from milliseconds to nanoseconds

Analysis:

I conducted two performance experiments:

In the first experiment, I continuously created points and called their public methods, measuring the number of points generated within various time intervals (from 5 to 25 seconds in steps of 5 seconds) as shown in Table 1. Additionally, I conducted a statistical analysis by recording the number of points created in 10 seconds over 20 runs and calculated the median, maximum, and minimum values, presenting the results in Table 2. Table 1 and 2 reveal that implementing Design 1 was quicker than Design 5. However, when it came to invoking methods, Design 5 was faster .

Moving on to the second experiment, I created different numbers of points (m) and measured the time (in milliseconds) required to both create and call these points, as indicated in Table 3.Furthermore, to conduct a basic statistical analysis, I measured the amount of time required to create 10 000 000 points over 20 runs and calculated the median, maximum, and minimum amount of time required (see Table 4). Similar to the first set of experiments, the results reveal that instantiating Design 1 was faster than Design 5, however when calling methods Design 5 was much faster.

From these experiments, I can conclude that Design 1 outperforms Design 5 in terms of rapid instance creation, making it an ideal choice for scenarios requiring quick instantiation of objects. On the other hand, Design 5 displays better performance in method calling, particularly noticeable in Table 1 and 3. Therefore, the choice between the two designs should be made based on the specific requirements of the application. If the primary concern is quick creation of objects, Design 1 is the preferred option. However, if efficient method calling is crucial, Design 5 offers a more optimal solution.

Exercise 3

Table 1 - Performance for Adding Elements

	Time taken to Add Elements (milliseconds)
ArrayList	2350
LinkedList	7585
Array	635

Table 2 - Performance for Calculating Sum of Elements

	Time taken to Calculate Sum of Elements (milliseconds)
ArrayList	126
LinkedList	1324
Array	53

The choice of a data structure depends significantly on its intended purpose. When adding elements, using an Array isn't ideal due to its fixed size, requiring prior knowledge of the exact data size. Although Arrays offer fast element addition with pre-allocated space, exceeding the allocated size necessitates creating a new, larger array and copying the elements, leading to inefficiency. An ArrayList resolves this problem, enabling quicker addition while maintaining greater flexibility than an Array. However, if insertion operations are necessary, a LinkedList would be a more suitable choice. As for calculating the sum of all elements, an Array or ArrayList would be preferable in terms of speed as they are indexed.

In conclusion, it is evident that using an Array would yield the best timing. Nevertheless, it's impractical to recommend a data structure solely based on the results of one experiment. Instead, its functionality/purpose should be taken into account for a decision. Therefore, opting for an ArrayList is more practical as it eliminates the inconvenient sizing aspect of an array and processes data much faster than a LinkedList.