SEG 2105
Assignment 1
300305189, 300303727

E26

	Pros	Cons
Design 1: Store one type of coordinates using a single pair of instance variables, with a flag indicating which type is stored.	- This design is relatively simple, making it easier to implement and understand, on the side of the user and programmer. - Extending this implementation is also made simpler due to the fact that there is only one type of coordinates. By adding more flags and types you can add more varieties of coordinates. - Only storing one pair of coordinates with a flag will expend less memory than having to store multiple coordinates.	- Only storing one type of coordinate at a time may be limiting if the program needs to utilize two types of coordinates together in the main method. It could be error-prone and cause inefficiency due to the constant converting between coordinate types. - This type of design also makes the code more difficult to read and tedious, as you are required to constantly change the flag to ensure which type of coordinate you are working with.
Design 2: Store polar coordinates only.	- Less memory used compared to design 1, 4 , and 5 due to one less method (convert polar to cartesian) - Computations involving polar coordinates will be a lot less complicated. - It is easier to retrieve polar coordinates.	- No checking for illegal argument exceptions in the constructor. - Designing PointCP in this way would restrict the way the class could be implemented, and terminate the functionality of many operations that utilize Cartesian coordinates. - Some methods will run slower.
Design 3: Store Cartesian coordinates only.	- Less memory used compared to design 1, 4 , and 5 - Computations involving cartesian coordinates will be much easier - It is easier to retrieve Cartesian coordinates. - Some methods will run quicker.	- No checking for illegal argument exceptions in the constructor. - Only storing Cartesian coordinates would limit many mathematical computations as you would need to convert the Cartesian coordinates to other coordinate pair types.

Design 4: Store both types of coordinates, using four instance variables.	- This design would be preferred if you were using the program to do complex computations, as this design would not require tedious conversion that could be prone to errors. - This design eliminates the need for flags and converting between types of coordinates, making the code easier to use. - It is necessary to create more instances (2 for each type of coordinate), making it less efficient than other designs that require creating fewer variables.	- The amount of memory used would increase due to the additional instance variables. - This would also make the code more complex and decrease the readability. Adding any other pairs of coordinates to the implementation would also become much more difficult using this design.
Design 5: Abstract superclass with designs 2 and 3 as subclasses.	- Using an abstract superclass, you can ensure that the subclasses adhere to conditions, as an interface creates a contract for the subclasses to follow. - One can reuse code several times in your program. - Each coordinate pair can be handled appropriately, as they are separated. This allows for specific requirements needed for a single type to be met.	- This design would increase the complexity of the code as it involves inheritance. - The amount of memory used would also increase as you would need to implement a superclass and two subclasses. - The efficiency when doing computations would depend largely on how the Abstract class and subclasses were implemented, rather than the design. - Because the code must be standardized, some of the code may not be required for it to be functional.

E28

The performance analysis was conducted by calculating the time required for each of the design's methods to calculate all of the randomized inputs of the cartesian and polar coordinates. These tests were performed 10 times, in which each design called each method 10,000,000 times. The millisecond runtime was then displayed for each method, which we indicated in the chart below. We utilized a for loop, which created a fixed number of instances that helped us test the designs. It generated instances with random coordinate types and random X/Y and Rho/Theta values.

The results of the performance analysis were within the scope of our original hypotheses. The getX() and the getY() methods took significantly less time to execute than the getRho() and the getTheta() in Design 5 sub 2. This trend continues for Design 5 sub 3, where getX() and getY()'s runtime is reduced, however while getTheta() becomes much larger while getRho() stays relatively the same. This is likely due to the complex nature of the getTheta() function, which involves many calculations. However, the getRho() performance differed from our original expectations. After inspecting the code for errors we concluded that it is likely due to the distinct Math functions used to calculate both values. The functions used in getTheta() must take a significantly longer time to run than the getRho() functions.

These changes in the execution time for the methods has largely to do with the fact that Design 5 sub 2 stores polar coordinates only and Design 5 sub 3 stores cartesian coordinates only. This means that in Design 5 sub 2, polar coordinates are simply returned, and in Design 5 sub 3, cartesian coordinates are simply returned.

The getX() and the getY() methods did not require large calculations, therefore they were able to run faster. Subsequently, the methods that implemented getX() and getY() were also able to run quickly as the calculations were not very complex. That is why the getDistance() and the rotatePoint() functions have a faster runtime in Design 5 sub 3, which only store Cartesian coordinates. The rotatePoint() is overall the slowest performing function due to the complicated equations.

E30

				Design 5					
Operations	Design 1			Design 2			Design 3		
	MIN	MED	MAX	MIN	MED	MAX	MIN	MED	MAX
getX()	7.0 ms 0.007 s	15.0 ms 0.015 s	249.0 ms 0.249 s	214.0 ms 0.214 s	235.5 ms 0.2355 s	249.0 ms 0.249 s	11.0 ms 0.011 s	14.5 ms 0.0145 s	23.0 ms 0.023 s
getY()	6.0 ms 0.006 s	94.0 ms 0.094 s	250.0 ms 0.25 s	209.0 ms 0.209 s	228.5 ms 0.2285 s	245.0 ms 0.245 s	10.0 ms 0.01 s	13.5 ms 0.0135 s	23.0 ms 0.023 s
getRho()	3.0 ms 0.003 s	5.0 ms 0.0145 s	21.0 ms 0.021 s	11.0 ms 0.011 s	13.0 ms 0.013 s	24.0 ms 0.024 s	11.0 ms 0.011 s	13.5 ms 0.0135 s	22.0 ms 0.022 s
getTheta()	6.0 ms 0.006 s	14.0 ms 0.014 s	245.0 ms 0.245 s	11.0 ms 0.011 s	14.0 ms 0.014 s	27.0 ms 0.027 s	1404.0 ms 1.404 s	1668.5 ms 1.6685 s	1695.0 ms 1.695 s
convertStorageToPolar()	3.0 ms 0.003 s	6.0 ms 0.006 s	7.0 ms 0.007 s	11.0 ms 0.011 s	16.0 ms 0.016 s	28.0 ms 0.028 s	1387.0 ms 1.387 s	1659.5 ms 1.6595 s	1736.0 ms 1.736 s
convertStorageToCartesian()	5.0 ms 0.005 s	5.0 ms 0.005 s	7.0 ms 0.007 s	442.0 ms 0.442 s	462.5 ms 0.4625 s	487.0 ms 0.487 s	11.0 ms 0.011 s	16.0 ms 0.016 s	23.0 ms 0.023 s
getDistance()	7.0 ms	391.5 ms	958.0 ms	885.0 ms	912.0 ms	978.0 ms	11.0 ms	14.5 ms	32.0 ms

	0.007 s	0.3915 s	0.958 s	0.885 s	0.912 s	0.978 s	0.011 s	0.0145 s	0.032 s
rotatePoint( )	842.0 ms 0.842 s	1191.5 ms 1.906 s	1593.0 ms 1.593 s	3736.0 ms 3.736 s	3769.0 ms 3.769 s	3809.0 ms 3.809 s	1864.0 ms 1.864 s	1893.5 ms 1.8935 s	1920.0 ms 1.92 s

Part 2:

Collection Size = 167699999	Array	ArrayList	Vector
Time taken for each collection to add elements	8092 ms - 8.092 s	1541 ms - 1.541 s	2260 ms- 2.260 s
Time taken for each collection to sum elements	271 ms - 0.271 s	273 ms - 0.273 s	544 ms - 0.544 s

From the above table, we have concluded that the Array, on average, is the slowest collection to add elements. The ArrayList performs the fastest, due to their ability to efficiently handle resizing operations. Vectors take slightly longer than ArrayLists, likely caused by the fact that they are synchronized, making ArrayLists more efficient when calculating the time taken for each collection to add elements to itself. The Array necessitates having its full size predefined before being populated with its contents, while the ArrayList and Vector are created by repeatedly adding items, and allowing them to dynamically grow.

When summing the elements, on average, the fastest collection was the Array. Arrays are very efficient when it comes to summing the elements because it has direct access to locations in the memory. The ArrayList performed very similarly to the Array, making the difference in their average time miniscule. The Vector took the longest to sum the elements, also likely due to their synchronization.

To make recommendations for designers, it is first necessary to understand its specific implementation requirements. Arrays are preferred when efficiency is a primary objective, namely if the collection size is known. Despite the increased execution time, however, the ArrayList and the Vectors may be the better choice for their specific performance and use. For example, ArrayLists are more equipped to store an abundance of elements. If the designers needed specifically to sum elements, the Array would again be the best option. The ArrayList performed well for both functionalities, therefore implementing the ArrayList may be the most advisable in instances where the both are necessary.