Kyle Labelle 300295594
Ethan Leu 300314618

https://github.com/SEG2105BC-uOttawa/assignment-1-assignment1-300314618_300295594

Part 1

E26.

Design	Simplicity of Code	Efficiency when creating instances	Efficiency when doing computations	Memory used
Design 1	More complex than designs 2-4 due to the need for methods to convert between cartesian and polar depending on what was stored and the need to track the current type with the typeCoord variable.	This design is more efficient than design 4, however it is less efficient than design 1 and 2. Design 1 needs 3 parameters to be initialized: xOrRho, yOrTheta and the type that is initialized. As this design has less parameters than design 4, it is more efficient than that design, but due to deesign 1 having more parameters than 2 and 3, it is less efficient than those in this regard.	When switching a lot between Polar and Cartesian, this design is less efficient than designs 2-4 as it needs to perform a calculation every time it switches. However, if it performs many calculations of one type before converting between cartesian and polar, this design is more efficient than design 2 and 3, but less efficient then 4.	This design uses more memory than 2 and 3 as it needs to store 3 variables: xOrRho, yOrTheta and the typeCoord to remember the current type.
Design 2	Simpler than Design 1. Can cut the methods convertStorageToPolar and convertStorageToCartesian and	This design is more efficient than design 1 and 4 for initializing instances as it only needs two	This design is more efficient than design 1 when there are many computations converting	This design uses less memory than designs 1 and 4 as it only needs to remember 2 variables: rho

	simplify the methods getRho and getTheta.	parameters, rho and theta.	between coordinate types, as it only needs to do a computation for converting to cartesian and not converting back to polar. It is less efficient when it has to do a lot of conversions to cartesian.	and theta
Design 3	Simpler than Design 1. Can cut the methods convertStorageToPolar and convertStorageToCartesian and simplify the methods getX and getY.	This design is more efficient than design 1 and 4 for initializing instances as it only needs two parameters, x and y.	This design is more efficient than design 1 when there are many computations converting between coordinate types, as it only needs to do a computation for converting to polar and not converting back to cartesian. It is less efficient when it has to do a lot of conversions to polar.	This design uses less memory than designs 1 and 4 as it only needs to remember 2 variables: x and y
Design 4	Simpler than designs 1-3. Design 4 does not need conversion methods to switch between cartesian and polar coordinates, making it more efficient than design 1, and it	This design is less efficient than designs 1,2 and 3 as the design takes 4 parameters to initialize, an x, y, theta, and rho, which is more parameters than the other designs.	This design is the most efficient for doing computations as it stores x,y, rho and theta and therefore does not need to do any calculations to convert between polar and cartesian	This design uses more memory than the others, as it needs to keep track of 4 variables: x, y, rho and theta.

	can simply return each type of coordinate, making it more efficient than designs 2 and 3.		coordinates.	
Design 5	Most complex design. This design utilizes abstract classes for its implementation, with three classes needed to accomplish all tasks: PointCP2, PointCP3 and PointCP5.	This design is as efficient as design 1 as it also needs three parameters: an xOrRho, yOrTheta and a typeCoord to determine if the value stored is in polar or cartesian	This design is also the most efficient, equally efficient as design 4 as it uses its subclasses to return values of x and y (from design 3) and rho and theta (from design 2) without needing to use any calculations	This design uses as much memory as design 2 and 3, as it only needs to store the xOrRho and yOrTheta variables in either of its subclasses.

E30.

Trial #	PointCP2 Type C (ms)	PointCP2 Type P (ms)	PointCP3 Type C (ms)	PointCP3 Type P (ms)	PointC (ms)	PointP (ms)
1	1795	1788	4666	3963	4640	1774
2	1792	1818	4854	3877	4814	1776
3	1808	1838	4800	4303	5230	1812
4	1876	1828	4800	4107	4810	1844
5	1878	1950	4739	3876	4745	1838
MIN	1792	1788	4666	3876	4640	1774
MAX	1878	1950	4854	4303	5230	1844
AVG	1829	1844	4771	4025	4847	1808

Analysis:
Every design was tested using 100,000,000 calls to getX(), getY(), getRho(), and getTheta() each. For every test that was conducted with polar coordinates stored, the results were significantly faster than the ones which stored cartesian. This is due to the fact that conversion from polar to cartesian is much less resource intensive than the other way around. As such, the designs where polar coordinates were stored could simply return the values, instead of converting them each iteration, thus having a higher performance. Our results differed slightly from our expected results in the question E26. We had predicted that design 2 and design 3 would have the same efficiency, however we had not taken into account that the calculation for the conversion of polar to cartesian coordinates is less intensive than cartesian to polar, causing PointCP2 to be more efficient than PointCP3.

Part 2

Trial #	ArrayList Construction (ms)	Vector Construction (ms)	Array Construction (ms)	ArrayList Iteration (ms)	Vector Iteration (ms)	Array Iteration (ms)
1	11178	9250	2724	2316	5529	12
2	15482	13206	2489	3394	10774	16
3	11152	8991	1852	3236	10820	0
4	11325	9933	1865	2723	9459	0
5	11676	9234	2233	2526	7422	8
AVG	12163	10123	2233	2839	8801	7

Analysis:

From the data achieved through the tests in part 2, we can draw the following conclusions. In both cases, arrays proved the most efficient by far, however they have the drawback of being a set size. To increase the size of an array, a new bigger one must be constructed, and all of these array constructions will stack up to become less efficient. Arrays are the solution if it is known beforehand the size the data structure will be. Vectors and ArrayLists have the advantage of being dynamic. According to our test results, ArrayLists construction for creating 2250 instances of arrays of size 1000000 averaged out to 12163 milliseconds, whereas the vector construction put under the same test averaged to 10123 milliseconds, giving Vectors the advantage in terms of construction efficiency. For iterations however, the ArrayLists under the same test gave a time of 2839 milliseconds compared to the vectors 8801 milliseconds. This gives ArrayLists the big advantage in efficiency over Vectors in terms of Iterations.