Question E26: Pros and cons for each one of the designs

	Pros	Cons
Design 1	-Flexible dealing with polar or cartesian coordinates -Class is more complex -Less memory used	-Slower operations when converting from either polar coordinates to cartesian coordinates or vice versa -Must specify what type of coordinate is used
Design 2	-Faster to compute operations with polar coordinates as no conversion is needed -Class is simple -Faster at creating instances as there is no need to check coordinate type -Less memory used	-Can only store polar coordinates -Slower to compute with cartesian coordinates as conversion is needed -Not flexible
Design 3	-Faster to compute operations with cartesian coordinates as no conversion is needed -Class is simple -Faster at creating instances as there is no need to check coordinate type -Less memory used	-Can only store cartesian coordinates -Slower to compute with polar coordinates as conversion is needed -Not flexible
Design 5	-Do not have to specify the type of coordinate is used (will depend on the instance) -Faster runtime than design 1 -Removes duplicate code for getDistance() method -Flexible	-An additional abstract class is required: more complexity, additional code leading to potential issues, -Potential misuse of the abstract method -More memory used

Question E28: Performance analysis between design 5 and design 1

Trial 1: 1 million instances

```
Design 1 Performance (average time per method call in ns):
getX: 30
getY: 30
getRho: 27
getTheta: 56
convertToCartesian: 34
convertToPolar: 74

Design 5 Performance (Polar subclass, average time per method call in ns):
getX: 35
getY: 33
getRho: 26
getTheta: 26
convertToCartesian: 40
convertToPolar: 26

Design 5 Performance (Cartesian subclass, average time per method call in ns):
getX: 25
getY: 25
getRho: 26
getTheta: 83
convertToCartesian: 25
convertToPolar: 78
```

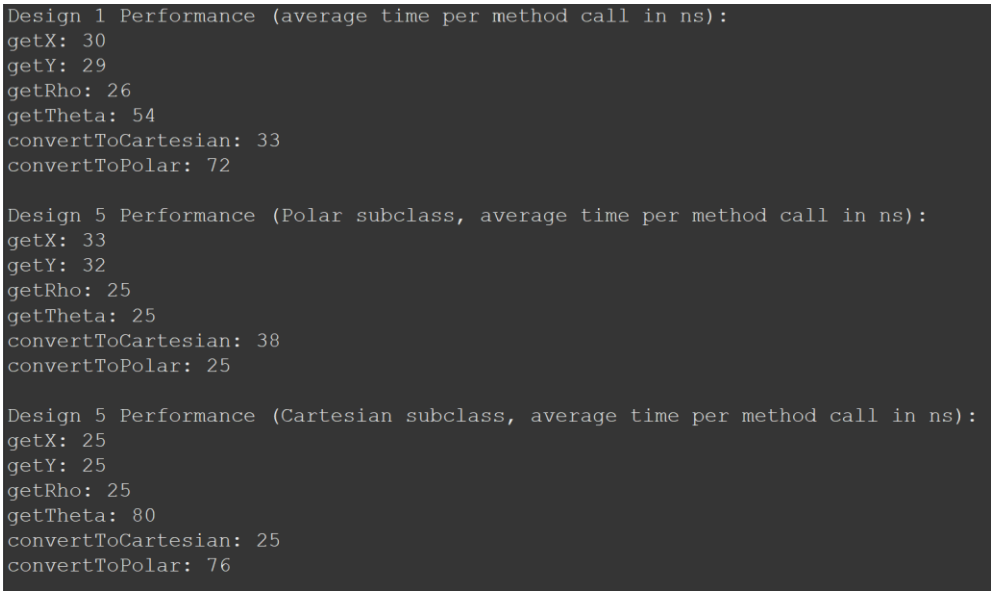
Trial 2: 1 million instances

```
Design 1 Performance (average time per method call in ns):
getX: 27
getY: 26
getRho: 23
getTheta: 48
convertToCartesian: 29
convertToPolar: 64

Design 5 Performance (Polar subclass, average time per method call in ns):
getX: 30
getY: 29
getRho: 23
getTheta: 23
convertToCartesian: 35
convertToPolar: 23

Design 5 Performance (Cartesian subclass, average time per method call in ns):
getX: 23
getY: 22
getRho: 23
getTheta: 73
convertToCartesian: 22
convertToPolar: 69
```

Trial 3: **1 million instances**



On average, after 1 million instances of each point class created and 3 trails here is the average for each method:

Design 1 average in ns:

1) getX(): 29
2) getY(): 28.3
3) getRho(): 25.3
4) getTheta(): 52.6
5) convertToCartesian(): 32
6) convertToPolar(): 70

Design 5 (Polar) average in ns:

1) getX(): 32.6
2) getY(): 29.3
3) getRho(): 24.6
4) getTheta(): 24.6
5) convertToCartesian(): 37.6
6) convertToPolar(): 24.6

Design 5 (Cartesian) average in ns:

1) getX(): 24.3
2) getY(): 24
3) getRho(): 24.6
4) getTheta(): 78.6
5) convertToCartesian(): 24
6) convertToPolar(): 74.3

Design 5 Total average in ns:

1) getX(): 28.45
2) getY(): 26.65
3) getRho(): 24.6
4) getTheta(): 51.6
5) convertToCartesian(): 30.8
6) convertToPolar(): 49.45

As we can see from the values above, we can see that design 5 has a faster average runtime than design 1. For the method getX(), design 5 had a faster runtime of 0.55 ns. For the method getY(), design 5 had a faster runtime of 1.65 ns. For the method getRho(), design 5 had a faster runtime of 0.7 ns. For the method getTheta(), design 5 had a faster runtime of 1 ns. For the method convertToCartesian(), design 5 had a faster runtime of 1.2 ns. Finally, the method convertToPolar(), design 5 had a faster runtime of 20.55. Thus, overall design 5 had a faster runtime than design 1 and verifies the hypothesis.

Question E29:

Found in GitHub, in the pointcp folder, and is labeled Test.

Question E30:

Performance Tests

The performance tests concerning designs 1, 5(polar), and 5(cartesian) were done by creating 1,000,000 random instances for each design and printing the difference between the start time and end time of each method. The 1,000,000 random instances ensure that the program runs for about 10 seconds, as instructed. The average runtime for each design is as follows:

	Design 1 (in ns)	Design 5: Cartesian (in ns)	Design 5: Polar (in ns)
getX	29	24.3	32.6
getY	28.3	24	29.3
getRho	25.3	24.6	24.6
getTheta	52.6	78.6	24.6
convertToCartesian	32	24	37.6
convertToPolar	70	74.3	24.6

The results indicate that design 5 (cartesian) runs its conversion to the other coordinate at an average speed of 74.3ns, while design 5(polar) runs its conversion at a speed of 37.6ns.

This is because the conversion from cartesian to polar coordinates requires more time-consuming methods such as the arctan operation, which are not used in the conversion from polar to cartesian coordinates.

The following code below, also found on the GitHub labeled mathPerformanceTest demonstrates the average runtime of the operations used within each PointCP class. The arctan operation is more time-consuming than the addition, exponential, cosine, and sine operations used in the polar to cartesian coordinate conversion. Thus, causing the PointCP3 to be slower than PointCP2.

```java
import java.util.function.BinaryOperator;

public class mathPerformanceTests {
    private static void test(String desc, BinaryOperator<Double> op, double a, double b, long startIter) {
        System.out.println("Running test: " + desc);
        long maxIter = startIter;
        long start;
        long elapsed;
        int loopCount = 0;
        final long MAX_ELAPSED_TIME_NS = 10_000_000; // 10 milliseconds in nanoseconds

        do {
            loopCount++;
            maxIter *= 2;
            start = System.nanoTime();
            for (long niter = 0; niter < maxIter; ++niter) {
                double res = op.apply(a, b);
            }
            elapsed = System.nanoTime() - start;
            System.out.println("Loop " + loopCount + ": elapsed time = " + elapsed + " ns");
        } while (elapsed < MAX_ELAPSED_TIME_NS && loopCount < 100); // Adjusted loop condition

        System.out.printf("%-15s/sec\t%g\n", desc, (maxIter * 1e9) / elapsed);
    }

    public static void main(String[] arg) {
        System.out.println("Starting performance tests...");

        test("Addition (double)", (Double a, Double b) -> a + b, 483902.7743, 42347.775, 10_000_000);
        test("Square", (Double a, Double b) -> a * a, 483902.7743, 42347.775, 10_000_000);
        test("Sine", (Double a, Double b) -> Math.sin(a), 0.5, 0, 1_000_000);
        test("Cosine", (Double a, Double b) -> Math.cos(a), 0.5, 0, 1_000_000);
        test("Arc Tangent", (Double a, Double b) -> Math.atan(a), 0.5, 0, 1_000_000);

        System.out.println("Performance tests completed.");
    }
}
```

Source Code: [Stack Overflow Code for Java Math Operators]

Another factor that design 1 is slower than design 5 is due to design 1 needing to use Boolean logic, such as “if” statements to determine what type of coordinate the method should compute. Whereas, design 5 does not need to evaluate any Boolean logic, as deepening on the point instance is what method will be executed.