CS419 Project 2: Multi-thread Monte Carlo Simulation for Finding the value of π

To be done individually.

Objectives:

1. Practice the basic use of the Executor framework (Java's multi-threading framework)
2. Compare the performance difference between the single-thread and multi-thread versions of the same program.
3. Gain understanding of the basic concept of the Monte Carlo method.

When it is due:

October 15, Friday, at 11:59pm.

What to submit:

1. Please submit the Java file that implements the multi-thread Monte Carlo simulation.
2. Please fill and submit the following table:

	Single-thread running time	Multi-thread running time	Single-thread π value	Multi-thread π value
200,000 total points	10 ms	19 ms	3.14592	3.1445
20M total points	134 ms	60 ms	3.1414424	3.1411706
2B total points	12260 ms	2605 ms	3.141619532	3.14156285

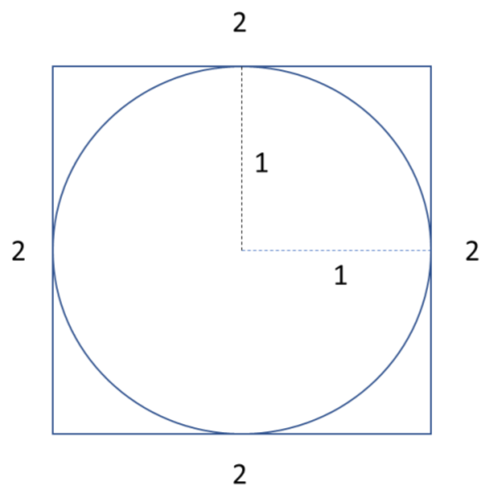
3. Please note: this is an individual assignment and everyone must submit it on Canvas. No teamwork is allowed.

Instructions:

1. Monte Carlo simulation is a random-sampling-based approach for finding approximate results to problems that are hard to solve directly. The accuracy of the approximation is typically dependent on the sampling size, so it is important to use sufficiently large number of samples, which may take a long time to run. Multiple threads can be used to speed up the simulation by performing multiple samplings in parallel, taking advantage of the multiple CPU cores. Monte Carlo simulation is widely used in physics, math, engineering, and finance. Please refer to the following Wiki page to learn more about this method.

https://en.wikipedia.org/wiki/Monte_Carlo_method

2. In this project, we will use the Monte Carlo method to approximate the value of π. The basic idea is the following:



   a. Imagine a circle with radius of 1. The area of the circle is π x 1 x 1 which is π. Imagine that the circle is tightly enclosed by a 2 x 2 square, as shown above. The area of the square is 4.
   b. We **randomly** drop multiple points onto the square. The points can end up in any position within the square, but many of them will also be within the boundary of the circle.
   c. We use the total number of points to represent the area of the square (which is 4) and use the number of points that are within the circle to represent the area of the circle (which is π). So, we can have the following equations:

$$\frac{area\ of\ the\ circle}{area\ of\ the\ square} = \frac{number\ of\ points\ within\ circle}{total\ number\ of\ points} = \frac{\pi}{4}$$

   It is easy to see that π can be approximated as:

$$\pi = 4 * \frac{number\ of\ points\ within\ circle}{total\ number\ of\ points}$$

   d. Clearly, this is only approximation. Its accuracy depends on the size of the samples (i.e., the number of points used in the simulation).

3. We provide you with the single-thread implementation of this simulation.
4. Please provide a multi-thread implementation of the same Monte Carlo simulation:

a. In your main thread, create an executor with four threads.
b. Each thread of the executor performs the Monte Carlo experiment **on a quarter of the graph, using a quarter of the total points**.
c. Each thread should return the number of points that are within its quarter of the circle.
d. In your main thread, calculate the π value using the method described in Step 2.c above.
e. Please note: You must use the Java Executor framework; you can reuse any code from our single-thread implementation.

Grading:

1. There is a total of 100 points for this project. You will receive 70 points for submitting a correct multi-thread implementation of the simulation (you will receive partial credit if your implementation is only partially correct). You will receive additional 30 points for completing and submitting the table using numbers produced from your simulations.
2. Please note: *you must use the Java Executor framework*. If your implementation is done in any other way, you will not receive any point for this project.

Additional Notes:

1. Java provides multiple ways to generate random numbers. In a multi-thread program, it is recommended that we use the *ThreadLocalRandom* class, as it is thread safe and each object is initialized with an internally generated seed (internal to each thread). Use of *ThreadLocalRandom* is particularly appropriate when multiple tasks use random numbers in parallel in thread pools. For more information about this class, please refer to the following Java doc: https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/ThreadLocalRandom.html