The learning outcome of this practical session is to rely on parallel streams to accelerate the computation of pi, a CPU-bound application.

1 Pi Computation

The value of $\pi$ is estimated by sampling random points in the square defined by the coordinates $(0, 0)$ and $(1, 1)$, and by counting how many are in the circle of radius 1 and which center is $(0, 0)$. This value divided by the number of samples and by 4 approximates $\pi$. This approach is called the Monte Carlo method: it consists in sampling $n$ points and averaging the results.

Even though the class `Random` can directly give a stream (see https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Random.html), it cannot be used directly to produce pairs of random values.

Write a stream implementation of this algorithm using `generate` and `Math.random`.

2 Parallel Version

To compare the performance of the sequential stream to the parallel one, it is necessary to select the number of samples $n$ for the measurements. The selected number of samples should be high enough to prevent the initialization from being preponderant, but small enough to avoid wasting time waiting for a result. Find a value $n$ that leads to a sequential execution of a few seconds (less than 5).

Measure precisely the time of the sequential execution of this implementation and its parallel execution to determine the speed-up (ratio of sequential time to parallel time).

Discuss and explain the measured performance relatively to the architecture and its potential for parallelization (use `lscpu` to determine the number of processors, cores and whether hyperthreading is enabled). Note that `Math.Random` is thread-safe.

Adapt the random generation by using `ThreadLocalRandom` instead of `Math.random` (see https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/ThreadLocalRandom.html). What is the new speed-up (perform each measurement multiple times, especially when the durations are short)?

3 Custom Spliterator

We will implement a spliterator that replaces both methods `generate` and `limit`: it will convey the size of the stream, which will improve the performance.

Implement a custom spliterator by redefining the methods `tryAdvance`, `trySplit`, `estimateSize`, and `characteristics`. The class `StreamSupport` can create a stream with a custom spliterator.

Compare the performance obtained with this custom to the performance with the direct approach with `generate`. 