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

The first two sections and the implementation of the Spliterator in the third section are essential.

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`.

```java
Stream.generate( () -> new double[]{Math.random(), Math.random()}
                .limit(n)
                .filter(x -> x[0] * x[0] + x[1] * x[1] < 1)
                .count() * 4. / n;
```

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).

Around 100 millions samples are necessary to have meaningful execution times.

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).

The measurements can be performed with `System.nanoTime()`.
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.

The method `Math.Random` is synchronized and a source of contention that slows down the computation.

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](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)?

```java
Supplier<double[]> supplier = () -> new double[] { ThreadLocalRandom.current().nextDouble(), ThreadLocalRandom.current().nextDouble() };
```

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 (`limit` will not introduce the SIZED flag).

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`.

```java
class SizedSpliterator<T> implements Spliterator<T> {
    private final long MIN_SIZE = 100;

    private long size;
    private Supplier<T> supplier;

    static <T> Stream<T> generateLimit(Supplier<T> supplier, long size) {
        return StreamSupport.stream(new SizedSpliterator<T>(supplier, size), false);
    }

    SizedSpliterator(Supplier<T> supplier, long size) {
        this.size = size;
        this.supplier = supplier;
    }
```
@Override
public boolean tryAdvance(Consumer<? super T> consumer) {
    if (size == 0)
        return false;

    T nextElement = supplier.get();
    --size;
    consumer.accept(nextElement);
    return true;
}

@override
public SizedSpliterator<T> trySplit() {
    if (size <= MIN_SIZE)
        return null;

    long half = size >> 1;
    size -= half;
    return new SizedSpliterator<T>(supplier, half);
}

@override
public long estimateSize() {
    return size;
}

@override
public int characteristics() {
    return DISTINCT | IMMUTABLE | SIZED | SUBSIZED | NONNULL;
}

SizedSpliterator.generateLimit(supplier, n)
 .parallel()
 .filter(x -> x[0] * x[0] + x[1] * x[1] < 1)
 .count() * 4. / n;