Learning objective: parallelize the reduce phase.
The parallelization of the reduce phase is illustrated with an algorithm to estimate the value of \( \pi \).

The first 5 questions are essential for the corresponding practical session.

Exercise 1: pi computation algorithm
What is the area of a circle with radius 1?

Its area is \( \pi \).

Let us draw a random point uniformly in the square defined by the coordinates \((0, 0)\) and \((1, 1)\).
What is the probability that this point is in the circle of radius 1 and which center is \((0, 0)\)?

Its probability is \( \pi/4 \).

By relying on the law of large numbers, specify an algorithm to estimate the value of \( \pi \) by drawing successively \( n \) random points.

Draw \( n \) random points uniformly distributed in the square defined by the coordinates \((0, 0)\) and \((1, 1)\). Test whether each point is in the circle of radius 1 and which center is \((0, 0)\) (the square root of the sum of the squares of x-value and y-value must be lower than 1). Count the number of such points and divide by \( n \) and by 4.

Exercise 2: stream formulation with \texttt{generate}
Propose a stream algorithm with the method \texttt{generate} to estimate the value of \( \pi \).

```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;
```

Exercise 3: stream formulation with \texttt{range}
The method \texttt{generate} leads to a stream that is unfortunately difficult to parallelize because it does not convey the size and the splitting step is thus inefficient.
Propose an alternative using the method \texttt{range}. 

Even though using `generate` followed directly by `limit` should be a sized stream, it is not always the case because a `filter` inserted between these two operations would prevent predicting the total number of elements to generate.

```java
LongStream.range(0L, n)
    .mapToObj(i -> new double[] { Math.random(), Math.random() })
    .filter(x -> x[0] * x[0] + x[1] * x[1] < 1)
    .count() * 4. / n;
```

Exercise 4: tryAdvance

In the following two exercises, we consider the implementation of a custom spliterator to split and iterate over a stream while retaining the size information, which is not the case by default with `generate` and `limit`.

We assume the spliterator is initialized with the appropriate supplier and size (`supplier` and `size`).

```java
class MySpliterator<T> implements Spliterator<T> {
  long size;
  Supplier<T> supplier;

  MySpliterator(Supplier<T> supplier, long size) {
    this.size = size;
    this.supplier = supplier;
  }
}
```

Let us consider the following simplified prototype: `boolean tryAdvance(Consumer<T> consumer)`. This function returns `true` if there are still elements in the stream, `false` otherwise. Its main purpose is to apply the consumer (with method `Consumer.accept(T)`) on a newly generated element (with method `Supplier.get()`) of the stream.

Propose an implementation of this method.

```java
boolean tryAdvance(Consumer<T> consumer) {
  if (size == 0)
    return false;
  T nextElement = supplier.get();
  size--;
  consumer.accept(nextElement);
  return true;
}
```

Exercise 5: trySplit

The method `trySplit` returns a new spliterator with half the elements or null if the stream cannot be split. Its prototype is `Spliterator<T> trySplit()`. Propose an implementation of this method.
`Spliterator<T> trySplit() {`n`    if (size <= 1L)`n`        return null;`n`    long half = size / 2;`n`    size -= half;`n`    return new MySpliterator<T>(supplier, half);`n`}

**Exercise 6: factorial**
Write a stream that computes the factorial of \( n \) in parallel.

```java
IntStream.rangeClosed(2, n)
          .parallel()
          .reduce(1, (x, y) -> x * y)
```

Discuss its parallelization compared to the parallelization of computing the sum of the first \( n \) numbers of the Fibonacci sequence.

On one hand, an `IntStream` is highly parallelizable.
On the other hand, the Fibonacci sequence requires a supplier with a mutable state that cannot be parallelized. In particular, with a parallel stream, the supplier must be thread-safe to prevent concurrent accesses and force the threads to access it in sequence.
If possible, it is thus better to design a supplier with an immutable state to achieve good parallelization.

**Exercise 7: matrix multiplication**
We assume that two matrices are given as in the following example:

```java
double[][] m1 = {{ 4, 8 }, { 0, 2 }, { 1, 6 }};
double[][] m2 = {{ 5, 2 }, { 9, 4 }};
```

In this case, the result of their product must be:

```java
double[][] res = {{ 92, 40 }, { 18, 8 }, { 59, 26 }}
```

Propose an implementation with external iterations (with `for` loops).

```java
double[][] res = new double[m1.length][m2[0].length];
for (int i = 0; i < m1.length)
    for (int j = 0; j < m2[0].length)
        for (int k = 0; k < m2.length)
            res[i][j] += m1[i][k] * m2[k][j]
```
Replace the most inner loop with a stream.

```java
res[i][j] = IntStream.range(0, m2.length)
    .mapToDouble(k -> m1[i][k] * m2[k][j])
    .sum()
```

Propose a complete stream implementation of the matrix multiplication that relies on the method `toArray`, which is a terminal operation that transforms the stream into an array.

```java
IntStream.range(0, m1.length)
    .mapToObj(i ->
        IntStream.range(0, m2[0].length)
            .mapToDouble(j ->
                IntStream.range(0, m2.length)
                    .mapToDouble(k -> m1[i][k] * m2[k][j])
                    .sum())
            .toArray())
    .toArray(double[]::new)
```

Remark that this code is not particularly easy to read. Using an external iteration, although not automatically parallelizable, is clearer. Streams should thus be used with caution to avoid degrading code clarity (the best solution is sometimes to do a mix of both external and internal iterations).

Propose and discuss multiple options to parallelize this stream. 

There are 3 streams that can be parallelized. Parallelizing the most inner stream leads to small tasks (fine grain parallelization). Parallelizing the most outer stream leads to larger tasks, which is likely to be more efficient because it minimizes the task creation overhead. However, it is necessary to measure actual performance.