The learning outcome of this practical session is to practice Java Streams with advanced collectors. The objective consists in partitioning numbers into prime and nonprime.

The first section and the supplier and accumulator of the second are essential.

1 Partitioning with a Standard Collector

Using a stream, write a *predicate* that tests whether a given candidate number is prime or not (from now on, external iterations with loops will be avoided).

1. The primality test consists in checking whether the candidate number is a multiple of any number between 1 and itself.

```java
public boolean isPrime(int candidate) {
    return IntStream.range(2, candidate)
        .noneMatch(i -> candidate % i == 0);
}
```

2. Write a method taking as argument an integer $n$ and partitioning the first $n$ natural numbers into prime and nonprime in an object of type $\text{Map}\langle\text{Boolean}, \text{List}\langle\text{Integer}\rangle\rangle$.

```java
public Map<Boolean, List<Integer>> partitionPrimes(int n) {
    return IntStream.rangeClosed(2, n).boxed()
        .collect(partitioningBy(candidate -> isPrime(candidate)));
}
```

2 Partitioning with a Custom Collector

This first solution has one drawback: for each candidate, we check whether it is a multiple of any of the numbers between 2 and some upper bound, whereas we are interesting in checking only if it is a multiple of any of the prime numbers found so far between 2 and some upper bound. We now want to generate a $\text{List}\langle\text{Integer}\rangle$ of all primes.

1. Rewrite the function testing the primality by considering two arguments: a list of prime numbers found so far and the current candidate number to be tested.
public boolean isPrime(List<Integer> primes, int candidate) {
    return primes.stream().noneMatch(i -> candidate % i == 0);
}

2. Let’s define a collector that will use this function to process the prime numbers. In public interface Collector<T, A, R>. What are T, A and R in our case? Refer to the official documentation of this interface (select the one specific to the version of Java that is being used).

    T is Integer and A and R are both List<Integer>. The working version is obtained with java -version. The official documentation is available on https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/Collector.html.

3. What are the supplier and the accumulator?

    public Supplier<List<Integer>> supplier() {
        return ArrayList<Integer>::new;
    }

    public BiConsumer<List<Integer>, Integer> accumulator() {
        return (acc, candidate) -> {
            if (isPrime(acc, candidate))
                acc.add(candidate);
        };
    }

4. What are the last three functions?

    The algorithm is inherently sequential, the combiner will never be called.

    public BinaryOperator<List<Integer>> combiner() {
        return (acc1, acc2) -> {
            throw new UnsupportedOperationException();
        };
    }

    public Function<List<Integer>, List<Integer>> finisher() {
        return Function.identity();
    }

    public Set<Characteristics> characteristics() {
        return Collections.unmodifiableSet(EnumSet.of(IDENTITY_FINISH));
    }

5. What is the final code to generate all primes from 2 to n?

    On obtient la fonction finale en implémentant l’interface Collector<Integer,List<Integer>, List<Integer>>:
```java
public List<Integer> partitionPrimesWithCustomCollector(int n) {
    return IntStream.rangeClosed(2, n).boxed()
        .collect(new PrimeNumbersCollector());
}
```

Alternatively, it is also possible to rely on the same collector algorithm in a more direct way as follows:

```java
IntStream.rangeClosed(2, n)
    .collect(ArrayList<Integer>::new, (acc, candidate) -> {
        if (isPrime(acc, candidate))
            acc.add(candidate);
    }, (acc1, acc2) -> {})
```

### 3 Optimization

Propose an optimization to speed up both predicates used above by improving the algorithm. To this end, remark that it is wasteful to test whether each number between 1 and the candidate is a divisor.

A non-prime number has necessarily a divisor that is less than or equal to its square root.

```java
public boolean isPrime(int candidate) {
    int candidateRoot = (int) Math.sqrt(candidate);
    return IntStream.rangeClosed(2, candidateRoot)
        .noneMatch(i -> candidate % i == 0);
}
```

```java
public boolean isPrime(List<Integer> primes, int candidate) {
    int candidateRoot = (int) Math.sqrt((double) candidate);
    return primes.stream()
        .takeWhile(i -> i <= candidateRoot)
        .noneMatch(i -> candidate % i == 0);
}
```