The learning outcome of this practical session is to understand the internals of a basic completable future.

1 Implementing CompletableFuture

Implement a simplified completable future that only consists of the following methods: `supplyAsync`, `thenApply` and `get` (follow the same method signature as the one from `CompletableFuture`). The solution can only use the following classes: `Future`, `ExecutorService` and `Supplier`. Moreover, to ensure that tasks are executed in sequence, you can rely on a single threaded executor such as the one provided by class `Executors`.

```java
class MyCompletableFuture<T> {
    Future<T> result;
    ExecutorService exec;

    private MyCompletableFuture(Future<T> result, ExecutorService exec) {
        this.result = result;
        this.exec = exec;
    }

    static <U> MyCompletableFuture<U> supplyAsync(Supplier<U> s) {
        var exec = Executors.newSingleThreadExecutor();
        var result = exec.submit(() -> s.get());
        return new MyCompletableFuture<U>(result, exec);
    }

    <U> MyCompletableFuture<U> thenApply(Function<T, U> o) {
        var newRes = exec.submit(() -> o.apply(result.get()));
        return new MyCompletableFuture<U>(newRes, exec);
    }

    T get() throws InterruptedException, ExecutionException {
        return result.get();
    }
}
```

2 Revision: Parallel Prime Numbers Implementation

This algorithm seen in the second practical session is inherently sequential.

Another approach consists in relying on the collect operation as follows: any time two lists must be combined, remove from the first each multiple of any value of the second and vice versa.
Implement the collector using the interface and the algorithm seen in the second practical session.

```java
public class PrimeNumbersCollector implements Collector<Integer, List<Integer>, List<Integer>> {
    @Override
    public Supplier<List<Integer>> supplier() {
        return ArrayList<Integer>::new;
    }

    public boolean isMultiple(int i, List<Integer> y) {
        return y.stream()
            .anyMatch(j -> i % j == 0);
    }

    public List<Integer> notMultiple(List<Integer> x, int j) {
        return x.stream()
            .filter(i -> i % j != 0)
            .collect(Collectors.toList());
    }

    @Override
    public BiConsumer<List<Integer>, Integer> accumulator() {
        return (x, j) -> {
            x.retainAll(notMultiple(x, j));
            if (!isMultiple(j, x))
                x.add(j);
        };
    }

    public List<Integer> notMultiple(List<Integer> x, List<Integer> y) {
        return x.stream()
            .filter(i -> !isMultiple(i, y))
            .collect(Collectors.toList());
    }

    @Override
    public BinaryOperator<List<Integer>> combiner() {
        return (x, y) -> {
            var list1 = notMultiple(x, y);
            var list2 = notMultiple(y, x);
            list1.addAll(list2);
            return list1;
        };
    }

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

    @Override
    public Set<Characteristics> characteristics() {
        return Collections.unmodifiableSet(EnumSet.of(
            Collector.Characteristics.IDENTITY_FINISH));
    }
}
```
IntStream.rangeClosed(2, n)
   .parallel()
   .boxed()
   .collect(new PrimeNumbersCollector())

Alternatively, the collector can be called as follows:

```java
var PNC = new PrimeNumbersCollector();
IntStream.rangeClosed(2, n)
   .parallel()
   .boxed()
   .collect(PNC.supplier(), PNC.accumulator(), PNC.combiner())
```

However, in this case, the combiner must be adjusted to be a `BiConsumer`. 