TD MCP – session 2 – Advanced Streams

Learning objective: manipulate Java Streams (data-processing operations and collectors).

The first 7 exercises are essential.

**Exercise 1: factorial**
Write a stream that computes the factorial of \( n \).

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

**Exercise 2: Fibonacci tuples series (iterate)**
The numbers in the following sequence are part of the Fibonacci series: 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... The first two numbers of the series are 0 and 1, and each subsequent number is the sum of the previous two.

The series of Fibonacci tuples is similar; you have a sequence of a number and its successor in the series: (0, 1), (1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21), ...

Generate the first 10 elements of the series of Fibonacci tuples using the `iterate` method.

The first problem is that the `iterate` method takes a `UnaryOperator<T>` as argument, and you need a stream of tuples such as (0, 1). We use an array of two elements to represent a tuple. For example, `new int[]{0, 1}` represents the first element of the Fibonacci series (0, 1). This will be the initial value of the `iterate` method:

```java
Stream.iterate(new int[]{0, 1}, ???)
    .limit(10)
    .map(t -> t[0])
    .forEach(System.out::println);
```

You need to figure out the `??` in the code. Remember that `iterate` will apply the given lambda successively.

```java
Stream.iterate(new int[]{0, 1}, t -> new int[]{t[1], t[0] + t[1]})
    .limit(10)
    .forEach(t -> System.out.println("" + t[0] + "," + t[1] + ")"));
```

`iterate` needs a lambda to specify the successor element. In the case of the tuple (3, 5) the successor is (5, 3+5) = (5, 8). The next one is (8, 5+8). The pattern is as follows: given a tuple, the successor is (t[1], t[0] + t[1]). This is what the following lambda specifies: \( t[1], t[0] + t[1] \rightarrow \text{new int[]} \{t[1]\} \). By running this code you’ll get the series (0, 1), (1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21), ...
Note that if you wanted to print the normal Fibonacci series, you could use a map to extract only the first element of each tuple:

```java
Stream.iterate(new int[]{0, 1}, t -> new int[]{t[1], t[0] + t[1]})
 .limit(10)
 .map(t -> t[0])
 .forEach(System.out::println);
```

This code will produce the Fibonacci series: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34...

**Exercise 3: Fibonacci tuples series (generate)**

Complete the following code to obtain the same result with `generate`.

```java
IntSupplier fib = new IntSupplier() {
    private int previous = 0;
    private int current = 1;
    public int getInt() {
        int oldPrevious = this.previous;
        int nextValue = this.previous + this.current;
        this.previous = this.current;
        this.current = nextValue;
        return oldPrevious;
    }
};
IntStream.generate(fib)
 .limit(10)
 .forEach(System.out::println);
```

**Exercise 4: joining strings with reducing**

In the following exercises, you have access to the methods `getType` and `getName` from class `Dish` that return both a `String`. The list of all dishes is in `menu`.

Which of the following statements using the `reducing` collector are valid replacements for this `joining` collector? Answer the question by carefully analyzing the type of each step and function.

```java
String shortMenu = menu.stream().map(Dish::getName).collect(joining());
```

1. ```java
   String shortMenu = menu.stream().map(Dish::getName)
   .collect(reducing((s1, s2) -> s1 + s2)).get();
```
Statements 1 and 3 are valid, whereas 2 does not compile.

1. This converts each dish in its name, as done by the original statement using the joining collector, and then reduces the resulting stream of strings using a String as accumulator and appending to it the names of the dishes one by one.

   The type analysis of this stream can be done as follows (starting from the most nested structures and then in order):
   - \((s1, s2) \rightarrow s1 + s2\) is a function \((String, String) \rightarrow String\)
   - `reducing(...)` is a `Collector<String,String,Optional<String>>`
   - `menu.stream().map(...)` is a `Optional<String>`
   - `is a Stream<String>`
   - `map(...).collect(...)` is a `Optional<String>`
   - `map(...).collect(...) is a Optional<String>`

2. This doesn’t compile because the one argument that `reducing` accepts is a `BinaryOperator<T>` that’s a `BiFunction<T,T,T>`. This means that it wants a function taking two arguments and returns a value of the same type, but the lambda expression used there has two dishes as arguments but returns a string.

3. This starts the reduction process with an empty string as the accumulator, and when traversing the stream of dishes, it converts each dish to its name and appends this name to the accumulator. Note that, as we mentioned, `reducing` doesn’t need the three arguments to return an `Optional` because in the case of an empty stream it can return a more meaningful value, which is the empty string used as the initial accumulator value.

   Note that even though statement 1 is a valid replacement for the joining collector, it is used here to demonstrate how the reducing one can be seen, at least conceptually, as a generalization of other collectors. Nevertheless, for all practical purposes we always suggest using the joining collector for both readability and performance reasons.

   To analyze the chained operations, we determine the type of the argument and returned value for each operation. `map` applies a function from a dish to a string and returns a stream of strings. `reducing` uses a binary operator from two strings to a string and returns an optional string. `get` returns a string.

Exercise 5: groupingBy

We would like to generate a list of dish names for each dish type. This would lead to the following output:

```
{MEAT=[pork, beef, chicken],
 FISH=[prawns, salmon],
 OTHER=[french fries, rice, season fruit, pizza]}
```

Write the code to achieve this objective.
Map<String, List<String>> tagsByType = menu.stream()
 .collect(groupingBy(Dish::getType,
    mapping(Dish::getName, toList())))

Exercise 6: flatMapping
A list of tags is associated to each dish name as follows (we assume that each possible dish appears in this map):

Map<String, List<String>> dishTags = new HashMap<>();
dishTags.put("pork",.asList("greasy", "salty"));
dishTags.put("beef", asList("salty", "roasted"));
dishTags.put("chicken", asList("fried", "crisp"));
dishTags.put("french fries", asList("greasy", "friended"));
dishTags.put("rice", asList("light", "natural"));
dishTags.put("season fruit", asList("fresh", "natural"));
dishTags.put("pizza", asList("tasty", "salty"));
dishTags.put("prawns", asList("tasty", "roasted"));
dishTags.put("salmon", asList("delicious", "fresh"));

We would like to generate a list of all unique tags of only the dishes contained in a given list menu using the flatMapping collector. This would lead to the following output:

[salty, greasy, roasted, fried, crisp, tasty, fresh, delicious, natural, light]

Write the code to achieve this objective.

List<String> tags = menu.stream()
 .collect(flatMapping(dish -> dishTags.get(dish.getName()).stream(),
    toList()))
 .stream()
 .distinct()
 .toList();

The type analysis of this stream can be done as follows (starting from the most nested structures and then in order):
• `dishTags.get(dish.getName())` is a `List<String>`

• `dish -> dishTags.get(...).stream()` is a function `Dish -> Stream<String>`

• `flatMap(..., toList())` is a `Collector<Dish,?,List<String>>`

For each dish, a list of tags is converted into a stream of strings. With `flatMap`, all streams are merged to form a single stream of tags. This stream of tags is then converted into a set with `toList`.

Other alternatives:

```
List<String> tags = menu.stream()
    .map(Dish::getName)
    .collect(flatMapping(name -> dishTags.get(name).stream(),
                        toSet()));
```

```
Set<String> tags = menu.stream()
    .collect(flatMapping(dish -> dishTags.get(dish.getName()).stream(),
                         toSet()));
```

```
List<String> tags = menu.stream()
    .map(Dish::getName)
    .flatMap(name -> dishTags.get(name).stream())
    .distinct()
    .toList();
```

**Exercise 7: collector composition**

We would like to generate a set of unique tags for each dish type. This would lead to the following output:

```
{MEAT=[salty, greasy, roasted, fried, crisp],
 FISH=[roasted, tasty, fresh, delicious],
 OTHER=[salty, greasy, natural, light, tasty, fresh, fried]}
```

Write the code to achieve this objective.

```
Map<String, Set<String>> tagsByType = menu.stream()
    .collect(groupingBy(Dish::getType,
        flatMapping(dish -> dishTags.get(dish.getName()).stream(),
                         toSet())));
```

To analyze the chained collectors, we determine the type of the input stream type, argument and returned value for each collector. `groupingBy` takes a stream of dishes and returns a map from a dish type to what is given by the following collector. `flatMap` takes a stream of dishes, applies a function from a dish to a stream of tags and returns what is given by the following collector. `toList` takes a stream of tags and returns a set of tags.

**Exercise 8: iterate**

Rewrite the following without the predicate in `iterate`:
Remember that `generate` and `iterate` require a short-circuiting operation such as `limit`, `findFirst`, `findAny`, `matchAll`, `matchNone` or `matchAny`. `filter` does not work because the stream is infinite.

Exercise 9: maximum collector alternative
Write an alternative to:

```java
Comparator<Dish> dishCaloriesComparator = Comparator.comparing(Dish::getCalories);
int maxCalories = menu.stream()
   .collect(maxBy(dishCaloriesComparator))
   .get().getCalories();
```

The type analysis of this stream can be done as follows (starting from the most nested structures and then in order):

- `Dish::getCalories` is a function `Dish -> int`
- `dishCaloriesComparator` is a `Comparator<Dish>`
- `maxBy(dishCaloriesComparator)` is a `Collector<Dish,?,Optional<Dish>>`
- `menu.stream().collect(...)` is a `Optional<Dish>`
- `menu.stream().collect(...).get()` is a `Dish`
- `menu.stream().collect(...).get().getCalories()` is a `int`

```java
int maxCalories =
   menu.stream().map(Dish::getCalories).reduce(Integer::max).get();
int maxCalories = menu.stream().mapToInt(Dish::getCalories).max().get();
```

Exercise 10: summing collector alternative
Write an alternative to:

```java
int totalCalories = menu.stream().collect(summingInt(Dish::getCalories));
```

```java
int totalCalories =
   menu.stream().map(Dish::getCalories).reduce(Integer::sum).get();
int totalCalories = menu.stream().mapToInt(Dish::getCalories).sum();
```
Exercise 11: using partitioningBy

Like the `groupingBy` collector, the `partitioningBy` collector can be used in combination with other collectors. In particular, it could be used with a second `partitioningBy` collector to achieve a multilevel partitioning. What will be the result of the following multilevel partitionings and what is the type of each returned value? Answer the question by carefully analyzing the type of each step and function.

1. `menu.stream() .collect(partitioningBy(Dish::isVegetarian, partitioningBy(d -> d.getCalories() > 500)));`

2. `menu.stream() .collect(partitioningBy(Dish::isVegetarian, partitioningBy(Dish::getType)));`

3. `menu.stream() .collect(partitioningBy(Dish::isVegetarian, counting()));`

1. This is a valid multilevel partitioning, producing the following two-level Map:

   ```
   { false={false=[chicken, prawns, salmon], true=[pork, beef]},
   true={false=[rice, season fruit], true=[french fries, pizza]}}
   ```

2. This will not compile because `partitioningBy` requires a predicate, a function returning a boolean. And the method reference `Dish::getType` can’t be used as a predicate.

3. This counts the number of items in each partition, resulting in the following Map:

   ```
   {false=5, true=4}
   ```

Exercise 12: simulating `takeWhile` in Java 8

The `takeWhile` method was introduced in Java 9, so unfortunately you cannot use this solution if you are still using Java 8. How could you work around this limitation and achieve something similar in Java 8?

Let us consider the following example:

```java
public static 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);
}
```

You could implement your own `takeWhile` method, which, given a sorted list and a predicate, returns the longest prefix of this list whose elements satisfy the predicate:
public static <A> List<A> takeWhile(List<A> list, Predicate<A> p) {
    int i = 0;
    for (A item : list) {
        if (!p.test(item))
            return list.subList(0, i);
        i++;
    }
    return list;
}

Using this method, you can rewrite the isPrime method and once again testing only the candidate prime against only the primes that aren’t greater than its square root:

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

Note that, unlike the one provided by the Streams API, this implementation of takeWhile is eager. When possible, always prefer the Java 9 Stream’s lazy version of takeWhile so it can be merged with the noneMatch operation.