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

We assume that a class `Dish` is defined with the methods: `getType`, `isVegetarian`, `getCalories` and `getName`. All dishes are stored into `menu`, which is a list.

The first 10 exercises are essential.

**Exercise 1: external vs. internal iteration**

Which stream operation would you use to refactor the following code?

```java
List<Dish> vegetarianDishes = new ArrayList<>();
for (Dish d: menu) {
  if (d.isVegetarian())
    vegetarianDishes.add(d);
}
```

What would be the resulting code?

You need to use the filter pattern:

```java
List<Dish> vegetarianDishes = menu.stream()
    .filter(Dish::isVegetarian)
    .toList();
```

Note that `Dish::isVegetarian` is a shorter alternative to `d -> d.isVegetarian()`.

**Exercise 2: external vs. internal iteration**

Which stream operation would you use to refactor the following code?

```java
List<String> highCaloricDishes = new ArrayList<>();
Iterator<String> iterator = menu.iterator();
while (iterator.hasNext()) {
  Dish dish = iterator.next();
  if (dish.getCalories() > 300)
    highCaloricDishes.add(d.getName());
}
```

What would be the resulting code?

You need to use the filter and map patterns:
List<String> highCaloricDish = menu.stream()
    .filter(dish -> dish.getCalories() > 300)
    .map(Dish::getName)
    .toList();

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

- dish -> dish.getCalories() > 300 is a predicate Dish -> boolean
- Dish::getName is a function Dish -> String
- menu.stream().filter(...) is a Stream<Dish>
- menu.stream().filter(...).map(...) is a Stream<String>
- menu.stream().filter(...).map(...).toList() is a List<String>

Exercise 3: mapping
Given a list of numbers, how would you return a list of the square of each number? For example, given [1, 2, 3, 4, 5] you should return [1, 4, 9, 16, 25].

You can solve this problem by using map with a lambda that takes a number and returns the square of the number:

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> squares = numbers.stream()
    .map(n -> n * n)
    .toList();

Exercise 4: filtering/slicing
Given a list of Dish with method getType, write the stream that puts in a list the first two meat dishes?

You can solve this problem by composing the methods filter and limit together and using toList() to convert the stream into a list as follows:

List<Dish> dishes = menu.stream()
    .filter(dish -> dish.getType() == "MEAT")
    .limit(2)
    .toList();

Exercise 5: nested streams
Write a stream that puts in a list only the integers from a first list that are contained in second one (like List::retainAll). For example, given the lists [1, 2, 3] and [1, 2, 4], the resulting list is [1, 2]. The solution must exhibit a nested stream.
For each element of the first list, all the values from the second list are considered with a stream.

```java
List<Integer> numbers1 = Arrays.asList(1, 2, 3);
List<Integer> numbers2 = Arrays.asList(1, 2, 4);
List<Integer> result = numbers1.stream()
    .filter(i -> numbers2.stream().anyMatch(j -> i == j))
    .toList();
```

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

- `numbers2.stream().anyMatch(j -> i == j)` is a `boolean`
- `i -> numbers2.stream().anyMatch(j -> i == j)` is a predicate `Integer -> boolean`
- `numbers1.stream().filter(...)` is a `Stream<Integer>`

**Exercise 6: flatMap**

Given a list of lists of numbers, how would you return a single flattened list of all nested numbers? For example, given a list `[[1, 2, 3], [2, 3, 4], [3, 4, 5]]`, you should return `[1, 2, 3, 2, 3, 4, 3, 4, 5]`.

```java
List<Integer> numbers1 = Arrays.asList(1, 2, 3);
List<Integer> numbers2 = Arrays.asList(2, 3, 4);
List<Integer> numbers3 = Arrays.asList(3, 4, 5);
List<List<Integer>> numbers = Arrays.asList(numbers1, numbers2, numbers3);

List<Integer> result = numbers.stream()
    .flatMap(List::stream)
    .toList();
```

The type analysis of this stream can be done as follows:

- `numbers.stream()` is a `Stream<List<Integer>>`
- `List::stream` is a function `List<T> -> Stream<T>` where `T` is an `Integer` here
- `numbers.stream().map(List::stream)` would be a a `Stream<Stream<Integer>>`
- `numbers.stream().flatMap(List::stream)` is a a `Stream<Integer>`

**Exercise 7: flatMap**

Given a list of strings, how would you return a list of all distinct sorted capitalized characters? For example, given two strings "dcb" and "abc", you should return ["A", "B", "C", "D"].

```java
String str1 = "dcb";
String str2 = "abc";
List<String> list = Arrays.asList(str1, str2);

List<String> result = list.stream()
    .map(String::toUpperCase)
    .toList();
```
Exercise 8: flatMap

Given two lists of numbers, how would you return all pairs of numbers? For example, given a list [1, 2, 3] and a list [3, 4] you should return [[1, 3], [1, 4], [2, 3], [2, 4], [3, 3], [3, 4]]. For simplicity, you can represent a pair as an array with two elements.

You could use two maps to iterate on the two lists and generate the pairs. But this would return a Stream<Stream<Integer[]>>. What you need to do is flatten the generated streams to obtain a Stream<Integer[]>. This is what flatMap is for:

```java
List<Integer> numbers1 = Arrays.asList(1, 2, 3);
List<Integer> numbers2 = Arrays.asList(3, 4);
List<int[]> pairs = numbers1.stream()
    .flatMap(i -> numbers2.stream()
        .map(j -> new int[]{i, j})
    )
    .toList();
```

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

- j -> new int[]{i, j} is a function Integer -> int[]
- numbers2.stream().map(j -> new int[]{i, j}) is a Stream<int[]>
- i -> ...map(...) is a function Integer -> Stream<int[]>
- numbers1.stream().flatMap(...) is a Stream<int[]>

How would you extend the previous example to return only pairs whose sum is divisible by 3?

You saw earlier that filter can be used with a predicate to filter elements from a stream. Because after the flatMap operation you have a stream of int[] that represent a pair, you only need a predicate to check if the sum is divisible by 3:

```java
List<Integer> numbers1 = Arrays.asList(1, 2, 3);
List<Integer> numbers2 = Arrays.asList(3, 4);
List<int[]> pairs = numbers1.stream()
    .flatMap(i ->
        numbers2.stream()
            .filter(j -> (i + j) % 3 == 0)
            .map(j -> new int[]{i, j})
    )
    .toList();
```
The result is \[[2, 4], [3, 3]\].

There are two alternatives where the filter is placed after the map or after the flatmap, but it is more efficient to filter early.

**Exercise 9: reducing**

How would you count the number of dishes in a stream using the `map` and `reduce` methods (i.e. without `count`)?

You can solve this problem by mapping each element of a stream into the number 1 and then summing them using `reduce`! This is equivalent to counting, in order, the number of elements in the stream:

```java
int count = menu.stream()
    .map(d -> 1)
    .reduce(0, (a, b) -> a + b);
```

A chain of `map` and `reduce` is commonly known as the map-reduce pattern, made famous by Google’s use of it for web searching because it can be easily parallelized. Note the built-in method `count` to count the number of elements in the stream:

```java
long count = menu.stream().count();
```

Another alternative: `menu.stream().reduce(0, (a, b) -> a + 1)`.

**Exercise 10: peek**

`list` is a `List<Integer>` containing all integers from 2 to 4.

What is the output when executing this stream?

```java
list.stream()
    .peek(System.out::print)
    .filter(i -> i % 2 == 0)
    .peek(System.out::print)
    .map(i -> -i)
    .peek(System.out::print)
    .sorted()
    .peek(System.out::print)
    .toList();
```

Only the call to `sorted` will wait for all elements to be processed.

Therefore, the correct sequence is: 2, 2, -2, 3, 4, -4, -4 and -2. We start with 2, 2 and -2 for the first element. Then, the second element, 3, is not kept. Then, the last element is processed: 4, 4 and -4. At this step, both remaining elements are sorted and finally printed: -4 and -2.

**Exercise 11: intermediate vs. terminal operations**

In the stream pipeline that follows, identify the intermediate and terminal operations?

```java
long count = menu.stream()
    .filter(dish -> dish.getCalories() > 300)
    .distinct()
    .limit(3)
    .count();
```
The last operation in the stream pipeline count returns a long, which is a non-stream value. It’s therefore a terminal operation. All previous operations, filter, distinct, limit, are connected and return a stream. They are therefore intermediate operations.

**Exercise 12: equivalence**
Is it possible to rewrite any instruction that uses `anyMatch` with either `noneMatch` or `allMatch`? When possible, propose how to rewrite these instructions given a predicate `pred()`.

Yes, the three functions are completely interchangeable (by adjusting the predicate):

```
anyMatch(x -> p(x))
!allMatch(x -> !p(x))
!noneMatch(x -> p(x))
```

**Exercise 13: short-circuiting**
Which of the following operations allows short-circuiting? `skip` - `findFirst` - `dropWhile` - `filter`

`findFirst`

**Exercise 14: intermediate vs. terminal operations**
Which of the following operations is terminal? `dropWhile` - `map` - `skip` - `toList`

`toList`