CHAPTER 10: STREAMS AND LAMBDA EXPRESSIONS

LEARNING OBJECTIVES

- Be able to explain what a lambda expression is
- Be able to define and apply functional interfaces
- Know important functional interfaces
- Be able to use lambdas to filter and process data
- Be able to explain what a stream is and how it works
- Know important create, intermediate and terminal operations and be able to use them

10.1 LAMBDA EXPRESSIONS

So-called anonymous inner classes help to pass a "piece of functionality" to a method as a parameter.

```
button.setOnAction(new EventHandler() {
  public void handle(Event event) {
    System.out.println("Button click!"); //Reaction to button click
  }
});
```

The example defines an anonymous (=unnamed) class that implements the `EventHandler` interface with exactly one `handle` method. The class is defined and instantiated at the same time. As a result, an object is passed to the `onAction` method, on which the `handle` method is then called when the button `button` is clicked.

LAMBDA EXPRESSIONS

Anonymous classes are cumbersome and difficult to read when passing a relatively small block of code.

So-called lambda expressions (lambdas for short) can be used as an alternative.

With a lambda, the button example can be written more compact like this:

```
button.setOnAction(event -> System.out.println("Button-Click!"));
```

A lambda is comparable to an anonymous method without a specification of a return type and triggered exceptions. A lambda consists of a parameter list followed by "->" and an expression or a statement block:

```
<parameter list> -> <expression> | <block>
```

LAMBDA PARAMETER LIST

The parameter list requires brackets if there is more than one parameter or if data types are specified. The parameters are to be separated by commas as usual, the data types are optional. If omitted, the compiler will derive them automatically from the interface that is being implemented (see section about *Functional Interfaces*).

```
(int a, int b)
(a, b)
```

LAMBDA EXPRESSION WITHOUT BLOCK

If an expression is used, the compiler implicitly adds a "return" so that the expression can be evaluated at runtime and returned as a result.

Example: p is a parameter of type person

```
p -> p.getGender() == Sex.MALE && p.getAge() >= 18 && p.getAge() <= 25
```

The expression returns true for men of age 18 to 25

LAMBDA EXPRESSION WITH BLOCK

A block can be used instead of an expression. For the previous lambda, we now need a `return` statement:

```
p -> {
  return p.getGender() == Sex.FEMALE &&
    p.getAge() >= 18 && p.getAge() <= 25;
}
```

The expression returns true for women of age 18 to 25

COMPARATOR WITH LAMBDA

Instead of a comparator object for comparing integer objects, the lambda `intComparator` is passed, which causes the `int` list to be sorted in descending order.

```
Comparator<Integer> intComparator = (int1, int2) -> int1-int2;
List<Integer> integers = Arrays.asList(1, 2, 13, 4, 15, 6, 17, 8, 19);
Collections.sort(integers, intComparator);
System.out.println(integers);
```

Alternatively, you could also write the lambda as:

```
Comparator<Integer> intComparator2 = (int1, int2) -> {
  if (int1 < int2)
    return 1;
  else if (int1 > int2)
    return -1;
  return 0;
};
```

FUNCTIONAL INTERFACES

Before further possible applications can be presented, the term **functional interface** must be clarified. A functional interface is an interface with exactly one method, such as the interface `Comparator<T>`, which requires the method *compare*. With Java 8, functional interfaces are introduced as a separate data type in order to be able to define lambda expressions as instances of functional interfaces (=objects that implement a functional interface).

Example of a functional interface:

```
@FunctionalInterface
interface MathOperation { int operation(int a, int b); }
```

The interface is a data type that corresponds to every lambda expression that accepts two int parameters and returns an int result. Example:

```
MathOperation addition = (int a, int b) -> a + b;
```

```
int operate(int a, int b, MathOperation mathOperation) {
  return mathOperation.operation(a, b);
}
```

The lambda matches MathOperation, so the following call is correct:

```
int result = operate(3, 5, addition);
```

THE PREDICATE INTERFACE

A predefined functional interface is *Predicate<T>* for checking conditions (= predicate) for an object of type T:

```
@FunctionalInterface
interface Predicate<T> {
    boolean test(T objectToBeTested);
}
```

Every object that implements Predicate therefore offers a test method to which an object of type T is passed and which returns true or false.

```
Predicate<Person> checkAge = p -> p.getAlter() >= 18;
```

`checkAge` *checks whether a given person is of legal age.*

THE CONSUMER INTERFACE

A consumer object provides a method accept(T objectToBeConsumed), which processes the passed object of type T (e.g. outputs it to the console):

```
@FunctionalInterface
interface Consumer<T> {
    void accept(T objectToBeConsumed);
}
```

Every object that implements Consumer therefore offers an accept method that is passed an object of type T and returns nothing.

```
Consumer<Person> printPerson = p -> System.out.println(p);
```

printPerson outputs the passed Person object to the console.

Please be aware: All functional interfaces are recommended to have an informative @FunctionalInterface annotation. This not only clearly communicates the purpose of this interface, but also allows a compiler to generate an error if the annotated interface does not satisfy the conditions.

FILTERING AND EDITING LISTS

Given a Person class, see class definition below. Now we would like to filter a list of persons according to criteria that should be flexibly defined. Then, we would like to execute an action on each element of the list. In the example, the list is filtered by persons of legal age and then output to the console.

```
class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name; this.age = age;
    }
    public String toString() {
        return "Person: name = " + name + ", age = " + age;
    }
}
```

The list of persons is defined as follows:

```
List<Person> persons = Arrays.asList(
  new Person("Anna", 19), new Person("Hugo", 24),
  new Person("Codie", 18), new Person("Susi", 14));
```

Now two lambdas in the processPersons method can be used to filter and then output the list:

```
public void processPersons(List<Person> list,
    Predicate<Person> tester, Consumer<Person> action) {
  for (Person p : list)
    if (tester.test(p))
      action.accept(p);
}
```

The processPersons method simply iterates over the list of persons, checks each person with the predicate and - if the predicate returns true - forwards it to the consumer.

```
Predicate<Person> checkAlter = p -> p.getAge() >= 18;
Consumer<Person> printPerson = p -> System.out.println(p);
processPersons(persons, checkAge, printPerson);
```

The great advantage of this procedure lies in the flexibility of `processPersons`. The filtering and processing can be changed as required using the two parameters!

Another possible application for lambdas is the use of lambdas in aggregate functions for processing streams. Please refer to the examples in section 10.2.

ETA CONVERSION

There are often situations in which a method is to be executed on an object instance using lambda expressions. Example:

```
List<String> list = Arrays.asList("apple", "kiwi", "pear");
list.forEach((str) -> System.out.println(str));
```

The example outputs each element by calling `System.out.println`.

Alternatively, a double colon can be used to pass the object instance and the method call to be made in a short form.

```
List<String> list = Arrays.asList("Apple", "Kiwi", "Pear");
list.forEach(String.out::println);
```

Above `String.out` denotes the reference to the `PrintStream` instance and println denotes the method to be called on that instance during the iteration.

In order for methods to be specified via the so called η conversion, only the signature and the return data type must match.

Experiment with the η-conversion

```
public class DoubleColon {
    public interface ExampleI {
        int abc(int a, int b);
    }
    public int calc(int a, int b) {
        return a + b;
    }
    public int doSomething(ExampleI ex) {
        return ex.abc(30, 12);
    }
    public static void main(String[] args) {
        DoubleColon example = new DoubleColon();
        int calc2 = example.doSomething(example::calc);
    }
}
```

- The example defines a method `int calc(int, int)`
- ... and an interface `ExampleI` with a method `int abc(int, int)`

Experiment with the η-conversion

```
public class DoubleColon {
    public interface ExampleI {
        int abc(int a, int b);
    }
    public int calc(int a, int b) {
        return a + b;
    }
    public int doSomething(ExampleI ex) {
        return ex.abc(30, 12);
    }
    public static void main(String[] args) {
        DoubleColon example = new DoubleColon();
        int calc2 = example.doSomething(example::calc);
    }
}
```

- The method `doSomething` calls the method `abc` on an instance that can be substituted on `ExampleI`
- Using η-conversion , the `example` is used and the method `calc` is called. Without η-conversion it would be: `example.doSomething((a, b) -> example.calc(a, b))`

EXERCISE: LAMBDAS

Which statements regarding the following code section are correct?

```
Arrays.sort(personArray, (p1, p2) -> p1.name.compareTo(p2.name));
Comparable<Student> comparator = s -> getRegNr() - s.getRegNr();
MathOperation op = (x, y) -> x % y;
int result = op.operation(4, 2);
EventHandler<ActionEvent> handler = (int x) -> { };
```

- [ ] A lambda expression defines a nameless (anonymous) method.
- [ ] *personArray* is sorted in descending order by person name.
- [ ] *result* has the value 0.
- [ ] *handler* is syntactically correct.
- [ ] *comparator* is a comparator for comparing student objects based on their matriculation number.

10.2 STREAMS

Before processing a collection (e.g. email to a list of persons) often a filter is used to extract parts of the collection with certain properties (e.g. all female persons).

The stream concept was introduced in Java 8 to optimise such processes and avoid the typical iterations. A stream is a kind of conveyor belt on which operations are applied to all elements of a collection, one by one.

A typical stream works as follows:

Data → Stream → Operation$_1$ → ... → Operation$_n$ → Result

Data → Stream → Operation$_1$ → ... → Operation$_n$ → Result

the part **Data → Stream** is called create operation

the part **Operation$_1$→, ..., →Operation$_n$** are intermediate operations

the part **→Result** is the terminal operation

Example

```
List<Person> persons = …;
List<Person> adults = persons.stream()  // create
    .filter(p -> p.getAge() >= 18)      // intermediate
    .collect(Collectors.toList());      // terminal
```

- The stream operation is used to create a stream object for persons.
- The filter method reduces the number of person objects to adults (age >= 18!)
  …
- … while the predefined toList method converts the resulting filtered stream back into a list.

CREATE OPERATIONS

Conversion Array → Stream:

```
Person[] persons = …;
Stream<Person> stream = Arrays.stream(persons);
```

Conversion Collection → Stream:

```
Collection<Person> people = new ArrayList<Person>();
Stream<Person> stream = people.stream();
```

INTERMEDIATE OPERATIONS

- `Stream<T> filter(Predicate<T> p):`
  Returns a stream of T objects that satisfy p
- `Stream<R> map(Function<T, R> f):`
  Takes an input stream of Type T and returns a stream of R objects; f performs the mapping from T to R
- `Stream<T> sorted(Comparator<T> c):`
  Returns stream sorted according to c

TERMINAL OPERATIONS

- `void forEach(Consumer<T> action);`
  Executes action on each element of the stream
- `Object[] toArray();`
  Returns an array with the objects of the stream
- `T reduce(T identity, BinaryOperator<T> op);`
  Reduction operation, see example on the following page

- `T min(Comparator<T> c):`
  Returns the minimum of the stream according to c
- `T max(Comparator<T> c):`
  Returns the maximum of the stream according to c
- `long count():`
  Returns the number of elements in the stream
- `T findFirst():`
  Yields the first element of the stream

EXAMPLE: PROCESSING A LIST OF PERSONS

```
public static void main(String[] args) {
    List<Person> people = Arrays.asList(
        new Person("Anna", 19), new Person("Hugo", 24),
        new Person("Codie", 18), new Person("Susi", 14));
    int sumAge = people.stream()
    .mapToInt(p -> p.getAge())
    .reduce(0, (x, y) -> x + y);
    double averageAge = (double) sumAge / people.size();
    System.out.printf("Number of people: %d, average age: %f\n",
        people.size(), averageAge);
}
```

- The list of people is first converted into a stream.
- The mapToInt method (see Java API) converts the person stream into an int stream using the age of each person.
- Then reduce performs a summation of the age values, starting at 0.
- Finally, the average age is calculated and displayed.

ADDITIONAL INFORMATION

Stream API: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

Collector API:
https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html

EXERCISE: STREAMS

Which statements are correct?

- [ ] Stream processing is impossible without lambda expressions.
- [ ] Terminal operations represent the end of assembly line processing. They do not provide a stream.
- [ ] Create operations create a stream object from other data structures (list, array, ...).
- [ ] Intermediate operations change a stream and return a stream.

10.3 EXERCISE 1

TIME STREAM

In this exercise a list of messages is given. Each message contains a Unix timestamp in milliseconds and a message text. A Unix timestamp indicates the current time as the number of milliseconds since January 1, 1970.

The goal is to process the message list with a stream so that entries that do not fit the chronological order are filtered out and the remaining messages are output with a readable date or time format.

Accordingly this...

```
1581697831661: Message 1
1581553746009: Message 2
1588854263313: Message 3
```

... should be output as:

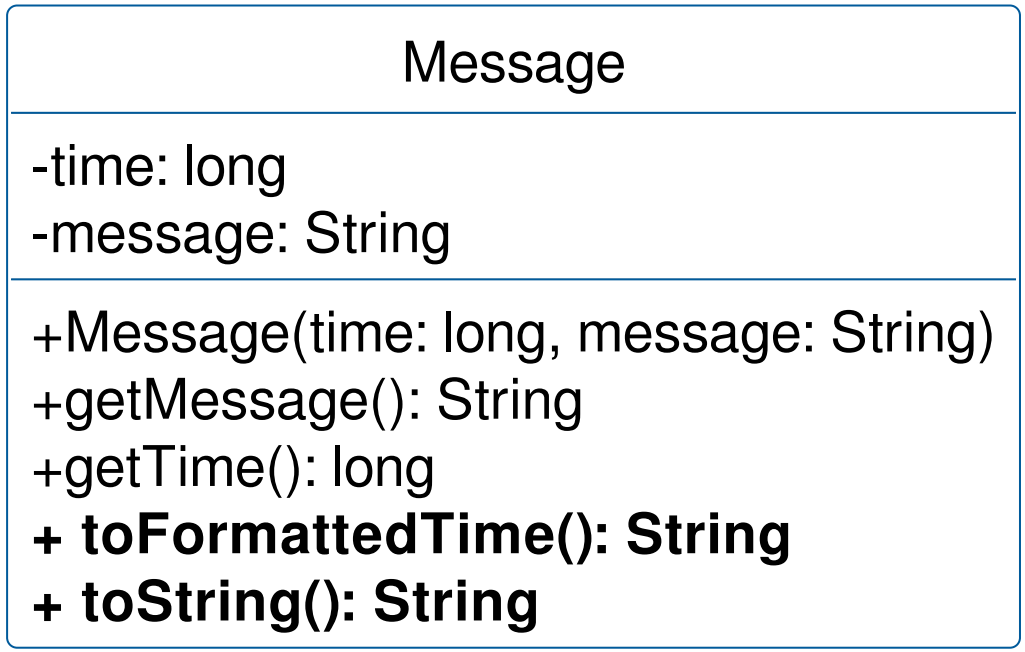
```
02/14/2020 05:30:31: Message 1
05/07/2020 02:24:23: Message 3
```

The timestamp of the 2nd message is smaller than the timestamp of the 1st message, so the 2nd message is skipped.

Unzip lambda-time.zip and import the folder as "Existing Maven Project".

STEP 1: EXTEND CLASS MESSAGE

Extend the inner class `Message` with the method `toFormattedTime` and `toString`.



Please be aware:

- The method `toFormattedTime` should generate a string from the `time` attribute.
  - Format: `<day>.<month>.<year> <hour>:<min>:<sec>`
  - Please be aware: Use the class `SimpleDateFormat`
- The `toString` method should return a string of the form `<formattedTime>: <message>` using the `toFormattedTime` method

STEP 2: STREAM OPERATIONS

The message list should be processed in the `handleList(List<String> messages)` method of the `Main` class. The list should be filtered and then output line by line.

The algorithm to be implemented is the following:

1. Break the message string into timestamps and message text
2. Generate a message from the two decomposition results
3. Ignore (=filter) the current message if it is not the first or if its timestamp is smaller than the last timestamp
4. Remember the timestamp for comparison with the next message
5. output all messages in the target format

Procedure for implementing the algorithm:

- Create a stream from the list
- Use the `map` operator and split the obtained string using the "`:`" (hint: `split()` method)
- Use the `map` operator and create a new instance of the `Message` class (hint: convert `String` to `Long` with `Long.valueOf()`)
- Use the `filter` operator to filter elements that do not fit in the order - use the static attribute `last` for this
- Use the `peek` operator and update the static attribute `last` with the current `Message` instance
- Use the terminal operation `forEach` to output the formatted entries

CHECK: TERMINAL-OPERATIONS

You used the following intermediate and terminal operations:

Map → Map → Filter → Peek → ForEach

What data type is processed by the consumer lambda in the terminal operation?

- [ ] String[]
- [ ] String
- [ ] Message
- [ ] boolean

CHECK: DATA TYPES DURING STREAM PROCESSING

You used the following intermediate operations:

Map → Map → Filter → Peek

What data type is passed on in the stream for each intermediate operation?

- ☐ String -> String -> String -> String
- ☐ String[] -> Message -> Message -> Message
- ☐ String[] -> Message -> boolean -> String

CHECK: RETURN TYPES OF THE INTERMEDIATE OPERATIONS

You used the following intermediate operations:

Map ➞ Map ➞ Filter ➞ Peek

What data type is returned by the respective lambda in the intermediate operations?

- ☐ String -> String -> String -> String
- ☐ String[] -> Message -> Message -> String
- ☐ String[] -> Message -> boolean -> void

10.4 EXERCISE 2

SHOPPING STREAM

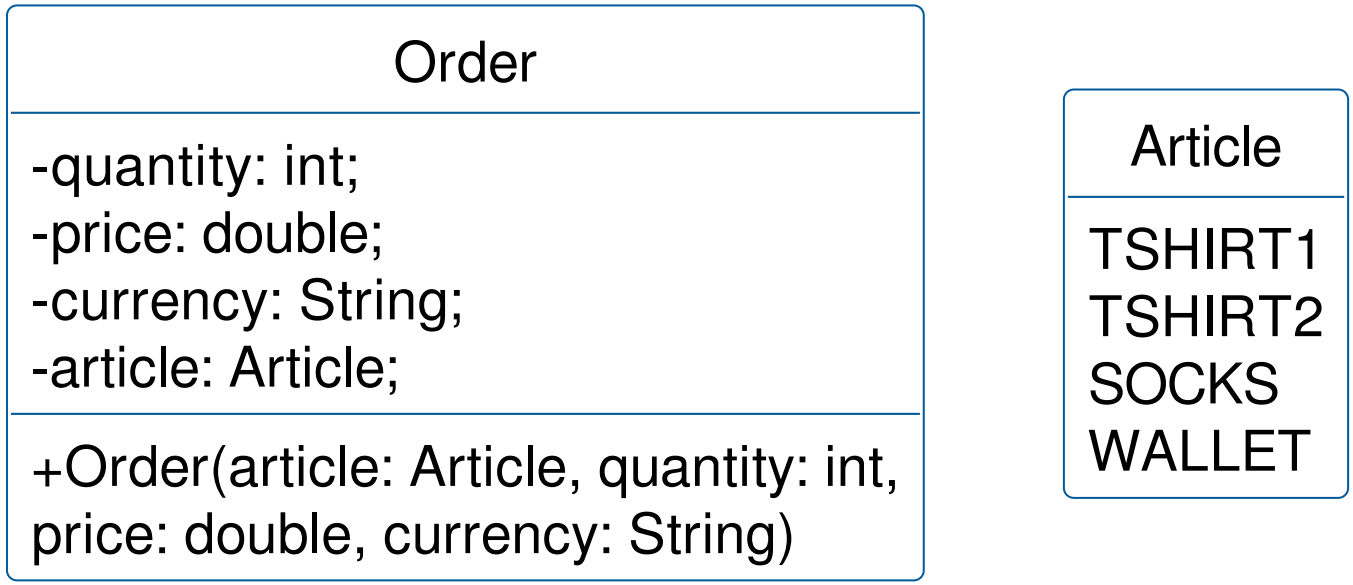
Given is a list of orders in a typical CSV (Comma Separated Values) format:

```
TSHIRT1;3;10.00;EUR
TSHIRT2;4;15.00;EUR
SOCKS;1;7.00;EUR
WALLET;4;5.0;EUR
WALLET2;7;6.5;EUR
```

This given order information must be checked and transferred to an instance of the Order class.

Unzip shopping-stream.zip and import the folder as "Existing Maven Project".

Given is a class Order and the enumeration type Article.



CHECKING STRINGS

Realize the methods `ShoppingStream.isInteger` and `ShoppingStream.isDouble`.

- The methods should check a passed string to see whether it can be converted into an integer or a double
- You can use the Integer.parseInt and Double.parseDouble methods for this
- These methods throw an exception NumberFormatException if an invalid string is passed
- Catch the exception and return `false`
- If no error occurs, `true` should be returned

ENUM TO MAP

Realize the `ShoppingStream.mapArticles` method

- The method should insert each instance of the enum type `Article` into a HashMap, with the name of the enum instance as key
- Use Stream API methods for this:
  - How to get an array of all enum types?
  - Convert the array to a stream
  - Use the terminal operation `collect`
  - Use the `Collectors.toMap`

PROCESSING ORDERS

Realize the `ShoppingStream.parseOrders` method

- Use Stream API for processing
- Split the string using the semicolon
- Check whether the resulting array contains 4 elements per entry
- Check whether the article is included in the article map
- Check whether the number is an integer (see isInteger)
- Check whether the price is a double (see isDouble)
- Use the Map operation to create an instance of the Order class from the string array
- Use the terminal operation `collect` to generate a list