CODEGYM

Mentor-supported training program from CodeGym

Java developer in 12 months

MODULE 2. JAVA CORE

Lesson #6

Lambda expressions

Lesson plan

- History of lambda expressions
- Lambda expressions in Java
- Functional methods
- Java 8: Functional programming
- Stream interface
- Method call chain
- Creating streams
- Data filtering
- ....

MODULE 2. JAVA CORE

History of lambda expressions

Interface is a kind of a class.
Heavily truncated version of a class, if I may say so. An interface, unlike a class, cannot have its own variables (except for static ones).
You also cannot create objects of the Interface type.

If a class implements an interface, it must implement internally those methods that were declared but not implemented inside the interface.

Sorting

In order to sort a collection of strings alphabetically, you can use an excellent Java method - Collections.sort(yourCollection);

Comparing elements in the process of sorting is carried out with th e help of the compareTo() method, which is available in all standard classes: Integer, String, ...

Method compareTo() of the Integer class compares values of two int egers, and the compareTo() method of the String class looks at the alphabetical order of strings.

CODEGYM

Alternative sorting

What if we want to sort strings not alphabetically, but by their lengths?
And what if we want to compare numbers in the descending order?

The Collections class has another sort() method for this, but with two parameters:

```
Collections.sort(yourCollection, comparator);
```

Where the Comparator is a special object which knows how
to compare objects in the process of sorting.

Comparator comes from the word "compare".

CODEGYM

Comparator interface

The type of the second parameter of the sort() method is — Comparator<T>

Where T is a type-parameter, the same as the type of the elements in the collection, and Comparator — an interface, which has a single method:

```
int compare(T obj1, T obj2);
```

In other words, the comparator object - is any object of a class which implements the Comparator interface.

```
public interface Comparator<Type>{
    public int compare(Type obj1, Type obj2);
}
```

The compare() method compares two parameters which are passed to it.
If the method returns a negative number, then **obj1 < obj2**.
If it returns a positive number, then **obj1 > obj2**.
If the method returns 0, then it is considered that **obj1 == obj2**.

Annonymous inner class

You can write the comparator code inside
the **main()** method, and the compiler will do the rest.

Example:

```
public class Solution{
    public static void main(String[] args){
        ArrayList<String> list = new ArrayList<String>();
        Collections.addAll(list, “Hello", “how", “are“, “you?”);
        Comparator<String> comparator = new Comparator<String>(){
            public int compare (String obj1, String obj2){
                return obj1.length() - obj2.length();
            }
        };
        Collections.sort(list, comparator);
    }
}
```

Lambda expressions in Java

Declaring a variable and creating an anonymous class can be written even shorter:

```
// Such code is called a lambda expression
Comparator<String> comparator = (String obj1, String obj2) -> {
    return obj1.length() - obj2.length();
};
```

Please note: when writing the lambda expression, we not only omitted the class name Comparat or<String>, but also the name of the int compare() method.

The compiler will have no problem defining the method, because lambda expressions can only be written for interfaces that have a single method.

The core of a lambda expression is the lambda operator, which is represented by the -> arrow. This operator divides the lambda expression into two parts: the left part contains a list of expression parameters, and the right part represents the actual body of the lambda expression, where all actions are performed on those declared in the left part.

Sorting (shortening the code)

```
//1
Comparator<String> comparator = (String obj1, String obj2) -> {
    return obj1.length() - obj2.length();
};
Collections.sort(list, comparator);
```

```
//2
Collections.sort(list, (String obj1, String obj2) -> {
            return obj1.length() - obj2.length();
        }
);
```

```
//3
Comparator<String> comparator = (obj1, obj2) ->
        obj1.length() - obj2.length();
Collections.sort(list, comparator);
```

```
//4
Collections.sort(list, (obj1, obj2) ->  obj1.length() — obj2.length() );
```

The compiler can determine that variables obj1 and obj2 are of the String type. Braces and the return operator can also be omitted, if your method has only one command.

Functional methods

If an interface has only one method, we can assign a value given by a lambda expression (lambda function) to a variable of this interface-type. Such interfaces are called functional interfaces (since lambda functions support was introduced in Java).

There's a Consumer<Type> interface in Java, which has the accept(Type **obj**) method. Why do we need this interface?

With the introduction of Java 8 the **forEach()** method was added to collections, which allows you to perform some action on each element of the collection. Here's where the functional **Consumer<T>** interface is used to transfer the action to the **forEach()** method.

Here's how you can output all elements of a collection on the screen:

```
ArrayList<String> list = new ArrayList<>();
Collections.addAll(list,  "Hello", "how", "are", "you?");

list.forEach( (s) -> System.out.println(s) );
```

Method reference

We can make our lambda expression even shorter.

```
list.forEach( (s) -> System.out.println(s) );
list.forEach( s -> System.out.println(s) );
```

We can only write it this way only when there's a single parameter.
If there are several parameters, then we need to use brackets.

If some action needs to be performed for each element of the list, and that action is a call to a function (such as println()), it makes more sense to simply pass a function as a parameter.

```
x -> object.method(x)
object::method
```

```
list.forEach( System.out::println );
```

CODEGYM

3 popular ways to pass a method reference

Reference to the object’s method

In order to pass a reference to the object’s method, we need to write:

```
object::method
```

This code is equivalent to:

```
x -> object.method(x)
```

Reference to a class method

In order to pass a reference to a static method, we need to write:

```
class::object
```

This code is equivalent to:

```
x -> class.method(x);
```

Constructor reference

A constructor in its behavior is somewhat similar to a static method of a class, so you can also pass a reference to it.

It looks like this:

```
class::new
```

Java 8: Functional programming

Powerful support for functional programming was introduced with the release of Java 8.

You could even say the long-awaited support for functional programming.

The code began to be written faster, although it became more difficult to read it.

Input-Output streams: chains of streams

Remember how we learned about the input-output streams:

```
InputStream, OutputStream, Reader, Writer
```

These streams can be organized into data processing chains:

```java
FileInputStream input = new FileInputStream("c:\\readme.txt");
InputStreamReader reader = new InputStreamReader(input);
BufferedReader buff = new BufferedReader(reader);

String text = buff.readLine();
```

When we call the buff.readLine() method, the following will happen:

1. **The BufferedReader object will call the read() method on the InputStreamReader object**
2. **The InputStreamReader object will call the read() method on the FileInputStream object**
3. **The FileInputStream object will start reading data from the file**

CODEGYM

Collections and streams

Starting from Java 8, it became possible to get a stream for reading data from collections (and not only from them).

Stream interface

In order to get the stream object from a collection, it is enough to call the stream() method. It looks something like this:

```
Stream<Type> name = collection.stream();
```

Where Type — is a type parameter, denoting the type of data to be used in the stream.

We can also get a stream out of an array, not just a collection:

```
Stream<Type> name = Arrays.stream(array);
```

CODEGYM

List of methods of the Stream type

Methods	Description
`Stream<T> of()`	Creates a stream from a set of objects
`Stream<T> generate()`	Generates a stream according to a given rule
`Stream<T> concat()`	Merges multiple streams together
`Stream<T> filter()`	Filters data: only passes data that matches the given rule
`Stream<T> distinct()`	Removes duplicates: does not pass data which has already been added
`Stream<T> sorted()`	Sorts data
`Stream<T> peek()`	Performs an action on each data
`Stream<T> limit(n)`	Truncates data after reaching a limit
`Stream<T> skip(n)`	Skips the first n data
`Stream<R> map()`	Converts data from one type to another
`Stream<R> flatMap()`	Converts data from one type to another
`boolean anyMatch()`	Checks that there is at least something in the stream data that matches the given rule
`boolean allMatch()`	Checks that all data in the stream matches the given rule
`boolean noneMatch()`	Checks that none of the data in the stream matches the given rule
`Optional<T> findFirst()`	Returns the first stream element which matches the given rule
`Optional<T> findAny()`	Returns any stream element which matches the given rule
`Optional<T> min()`	Returns the smallest element of the data stream
`Optional<T> max()`	Returns the largest element of the data stream
`long count()`	Returns the number of elements in the data stream
`R collect()`	Reads all data from the stream and returns it as a collection

Intermediate and terminal Stream methods

Not all methods provided in the table above return a Stream. It is because methods of the Stream class can be divided into intermediate (non-terminal) and final (terminal).

Intermediate methods
Intermediate methods return an object, which implements the Stream interface, and they can be lined up into a chain of calls.

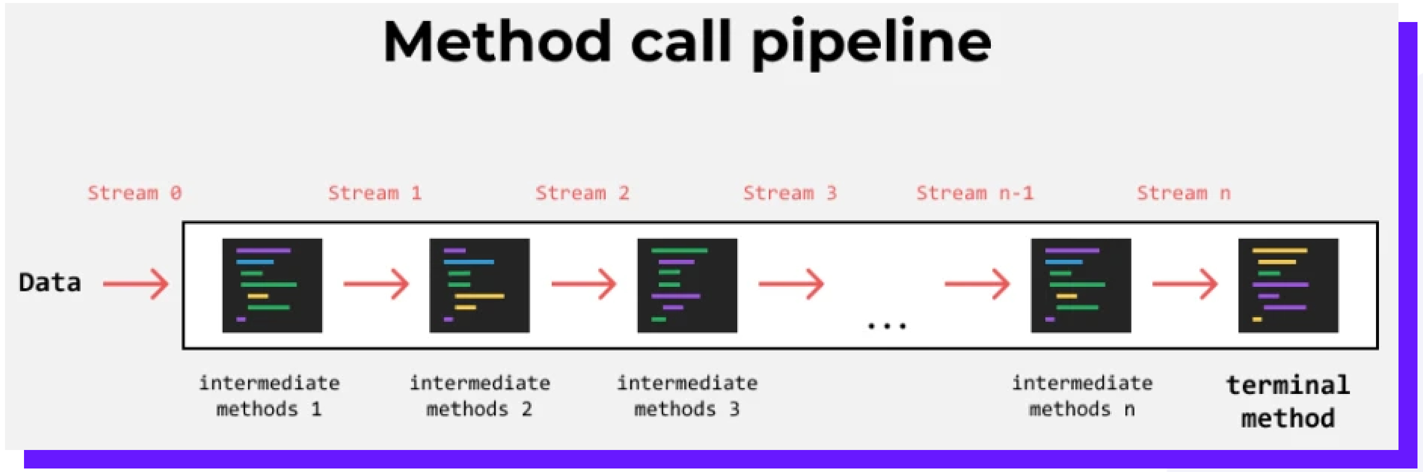
Final methods
Final methods return a value whose type is different to the Stream type.

CODEGYM

Chain of method calls

You can build chains of calls from any number of intermediate methods and then call one final method at the end.

This approach allows you to implement quite a complex logic, while increasing the readability of the code.



CODEGYM

Creating streams

Method Stream<T>.of(T obj)
This of() method creates a stream consisting of one element.

Method Stream<T> Stream.of(T obj1, T obj2, T obj3, ...)
This of() method creates a stream consisting of passed elements.

Method Stream<T> Stream.generate(Supplier<T> obj)
The generate() method allows to set a rule, using which the next element of the stream will be generated when it is requested. For example, you can provide a random number each time.

Method Stream<T> Stream.concat(Stream<T> a, Stream<T> b)
The concat() method merges two passed streams into one. When reading data, the data from the first stream will be read first, and then from the second one.

CODEGYM

Filtering data

Method Stream<T> filter(Predicate<T>)
This method returns a new data stream that filters the data from the source stream according to the passed rule. The method must be called on an object of type Stream<T>.

Method Stream<T> sorted(Comparator<T>)
This method returns a new data stream that sorts the data from the source stream. As a parameter, you can pass a comparator that will set the rules for comparing two elements of the data stream.

Method Stream<T> distinct()
This method returns a new data stream that contains only the unique data from the source data stream.

CODEGYM

Method Stream<T> peek(Consumer<T>)

This method returns a new data stream, although the data in it is the same as in the source stream. But when the next element from the stream is requested, the function that you passed to the peek() method is called for it.

Method Stream<T> limit(int n)

This method returns a new data stream that contains only the first n data from the source data stream. All other data is discarded.

Method Stream<T> skip(int n)

This method returns a new data stream that contains all the same data as the source stream, but skips (ignores) the first n data.

Data conversion

The Stream<T> class has a method, which allows you to convert data from one type to another. This method is called map().

It also returns a stream Stream<R>, but with elements of a new type. As a parameter, you need to pass a function to the map() method, that converts one data type to another.

```
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5);
Stream<String> stream2 = stream.map((x) -> String.valueOf(x));
```

CODEGYM

Data checks

Method boolean anyMatch(condition)
This method checks that there is at least one element in the stream that satisfies the rule that is passed to the method. If there is such an element, the method returns true, otherwise it returns false.

Method boolean allMatch(condition)
This method checks that all elements in the stream match the rule. The rule is passed to the method as a parameter.

Method boolean noneMatch(condition)
The noneMatch() method checks that there are no elements in the stream that match the passed rule. This method is the exact opposite to the anyMatch() method.

Utility classes: Optional class

The purpose of this class is simply to store an object T (a reference to an object of type T).

The reference to the object inside the Optional<T> class can be null.

Its code looks something like this:

```java
class Optional<T> {
    private final T value;
    private Optional() { this.value = null;}
    private Optional(value) { this.value = value;}
    public static <T> Optional<T> of(T value) {
        return new Optional<T>(value);
    }

    public boolean isPresent() {
        return value != null;
    }

    public boolean isEmpty() {
        return value == null;
    }

    public T get() {
        if (value == null) {
            throw new NoSuchElementException();
        }
        return value;
    }

    public T orElse(T other) {
        return value != null ? value : other;
    }

    public T orElseThrow() {
        if (value == null) {
            throw new NoSuchElementException();
        }
        return value;
    }
}
```

Roughly speaking, the Optional class allows you to write “prettier” null checks and actions, in case the Optional object is null stored inside.

Elements search

Method Optional<T> findFirst()
The findFirst() method simply returns the first element of a stream and that's it — this completes its work.

More interestingly, this method does not return an object of type T, but its wrapper - an object of type Optional<T>. This is to ensure that the method never returns null if it doesn't find an object.

Method Optional<T> findAny()
The findAny() method returns any element of a stream and then ends its work. This method is comparable to the findFirst() method, but for streams that are processed in parallel.

Method Optional<T> min(Comparator<T>)
The min() method compares all elements of the stream using the comparator object and returns the minimum element. The most convenient way to define a comparator object is with a lambda function.

Method Optional<T> max(Comparator<T>)
The max() method compares all elements of the stream using the comparator object and returns the maximum element. The most convenient way to define a comparator object is with a lambda function.

Collecting elements

The **collect()** method is used to change from streams to the more familiar collections — **List<T>, Set<T>, Map<T, R>** and others.

We need to pass a special object in the **collect()** method — a collector. This object reads all data from the stream, converts it to a specific collection and returns it. After that, the same collection is returned by the collect() method.

The Collectors class has several static methods that return ready-made collector objects for all occasions. There are several dozen of them, but we will go through the most basic ones:

`toList()`	An Object which converts a stream into a list — List<T>
`toSet()`	An Object which converts a stream into a set — Set<T>
`toMap()`	An Object which converts a stream into a map — Map<K, V>
`joining()`	Joins elements of a stream into a single string
`mapping()`	Converts stream elements into a map - Map<K, V>
`groupingBy()`	Groups elements and returns a map - Map <K, V>

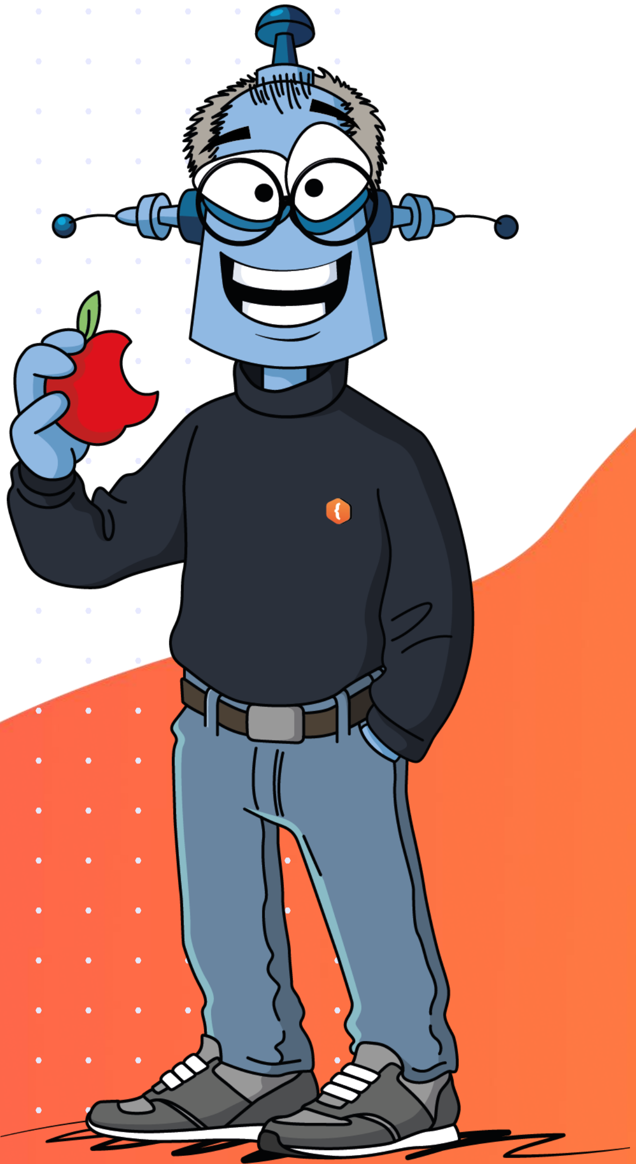
CODEGYM

Homework

Module 2

Level 6. Lambda functions

Questions and Answers



MODULE 2. JAVA CORE