Java Andvanced – lesson 2 – Advanced features

Wim Wiltenburg

(wim.wiltenburg@inholland.nl)

Recap week 1

After reviewing the submitted assignments, I have a couple of suggestions:

1. An endpoint must be clearly named and following conventions
2. Start your project as clean as possible
3. When you share your code to git, and I do comment, it will be in the form of an issue. Feedback on that issue is appreciated

First: an exercise

Write a small program that:

- Contains a list of numbers 1 to 10
- Takes all even numbers from the list
- Calculates their squares
- Sums them up
- Prints out the result

Java 8 up close and personal

- Java 8 came with a ton of new features that make the life of a developer a lot easier (and sometimes a lot harder)
  - Functional Style Progamming
  - Stream API
  - Time API
  - Java NIO.2
  - Other improvements like:
    - Comparator Interface
    - min(), max() and sum() method in Integer, Long and Double
    - etc.

Functional Style Programming

In Functional Programming data is handled by the evaluation of functions and expressions

It was a longstanding wish from the Java Community to add functional programming to Java. Java 8 fullfilled this wish.

Functional Programming is based on the principles of lambda calculus (λ)

In OO this type of programming would typically be handled by anonymous inner classes.

What is a function

Definition: a function is a process that assigns **each element** of a set X to a **single element** of set Y

In other words: the same input results in the same output.

Example: $f(x \in \mathbb{N}) = x^2$

f(1) -> 1

f(2) -> 4

In **higher order functions** you can pass functions to a function, or return a function from a function.

Functional Style vs OO

Is functional style programming just “syntax sugar”?

- Use functions
- Describe what to do
- Concise
- Contains no state
- Easily testable
- Idempotent by nature
- Easily maintained
- Pass data to function
- Pass function to function

- Use anonymouns inner class
- Describe how to do it
- Verbose – lot of boiler plate code
- Can have side effects
- Testable when well written
- Idempotent when well written
- Difficult to maintain
- Pass objects to method

Lambda expressions

Any normal function consists of 4 elements:

1. Name
2. Parameter List
3. Body
4. Return type

Coding example: Sorting guitars:

Comparable vs Comparator

Lambda expressions 2

Takes parameters and returns value

Without optional parts:

```
Collections.sort(guitars, (g1, g2) -> g1.getPrice() - g2.getPrice());
```

With optional parts:

```
Collections.sort(guitars, (g1, g2) -> {
        return g1.getPrice() - g2.getPrice();
    });
```

Functional Interface

To use functional programming in Java you need to use a Functional Interface:

Definition: **A Functional Interface is an interface that contains exactly one abstract method**.

Example:

```
public interface Swim {
    void stroke();
}
```

You can use @FunctionalInterface annotation as a contract. The code will not compile if it doesn't comply to the Functional Interface definition

Will these interfaces compile?

```
@FunctionalInterface
public interface Swim {
    void stroke();
    void crawl();
}

@FunctionalInterface
public interface Swim {
    void stroke();
    default void crawl() {
        System.out.println(“Crawling”);
    }
}

@FunctionalInterface
public interface Swim {
    void stroke();
    static void jump() {
        System.out.println(“Only when the cotton is high”);
    }
}
```

Built-in Functional Interfaces

java.util.function

- Predicate<T> boolean test(T t)
  Tests if a certain T is true or false
- Function<T,R> R apply(T t)
  Applies function on T to return R
- BiFunction<T,U,R> R apply(T t, U u)
  Applies function on T and U to return R
- Consumer<T> void accept(T t)
  Performs operation on T
- Supplier<T> T get()
  Gets result T

Predicate<T>

The one abstract method test(T t) produces a boolean.

Example: Does it start with…

```
Predicate<String> p = s -> s.startsWith("A");
 // Lambda expression. Read: for every String s does s start with “A”

boolean b1 = p.test("Alphabet");
boolean b2 = p.test("Beta Studies");

System.out.println(b1 + ", " + b2 );

Result:

true, false
```

Built-in Functional Interfaces

java.util.function

- Predicate<T> boolean test(T t)
  Tests if a certain T is true or false
- Function<T,R> R apply(T t)
  Applies function on T to return R
- BiFunction<T,U,R> R apply(T t, U u)
  Applies function on T and U to return R
- Consumer<T> void accept(T t)
  Performs operation on T
- Supplier<T> T get()
  Gets result T

Function<T,R>

The one abstract method apply(T t) returns an R when function applied to T

Example: Convert String to its size

```
Function<String, Integer> stringToSize = s -> s.length();
String aString = ”This String";
int size = stringToSize.apply(aString);
System.out.println(aString + ” has size: " + size);

Result:
This String has size: 11
```

When T, and R are the same type, it’s better do use UnaryOperator<T>

Built-in Functional Interfaces

java.util.function

- Predicate<T> boolean test(T t)
  Tests if a certain T is true or false
- Function<T,R> R apply(T t)
  Applies function on T to return R
- BiFunction<T,U,R> R apply(T t, U u)
  Applies function on T and U to return R
- Consumer<T> void accept(T t)
  Performs operation on T
- Supplier<T> T get()
  Gets result T

BiFunction<T,U,R>

The one abstract method apply(T t, U u) returns an R when function applied to T and U.

Example: concatenate a string

```
BiFunction<String, String, String> addStrings = (s1, s2) -> s1 + s2;
String one = "1";
String two = "2";
String twelve = addStrings.apply(one, two);
System.out.println(”Result = " + twelve);
```

```
Result:
Result = 12
```

When T, U and R are the same type, it's better do use BinaryOperator<T>

Built-in Functional Interfaces

java.util.function

- Predicate<T> boolean test(T t)
  Tests if a certain T is true or false
- Function<T,R> R apply(T t)
  Applies function on T to return R
- BiFunction<T,U,R> R apply(T t, U u)
  Applies function on T and U to return R
- Consumer<T> void accept(T t)
  Performs operation on T
- Supplier<T> T get()
  Gets result T

Consumer<T>

The one abstract method accept(T t) accepts input, and produces nothing. Its return type is therefore void

Example: printing a string

```
public class App
{
    public static void main( String[] args ) {
          Consumer<String> printer = App::printString;
          // method reference. Java knows the type of parameter it takes
          printer.accept("Hello World!");
    }

    private static void printString(String s) {
          System.out.println(s);
    }
}

Result:
Hello World!
```

Built-in Functional Interfaces

java.util.function

- Predicate<T> boolean test(T t)
  Tests if a certain T is true or false
- Function<T,R> R apply(T t)
  Applies function on T to return R
- BiFunction<T,U,R> R apply(T t, U u)
  Applies function on T and U to return R
- Consumer<T> void accept(T t)
  Performs operation on T
- Supplier<T> T get()
  Gets result T

Supplier<T>

The one abstract method get() produces a T when executed.
Example: create a new ArrayList<String>

```
Supplier<ArrayList<String>> supplier = ArrayList::new;
List<String> stringList = supplier.get();
stringList.add("String 1");
System.out.println(stringList);
```

```
Result:
[String 1]
```

Passing a lambda as parameter

You can pass a lambda expression as a parameter to a method. A function is an object and can be passed:

```
public class App
{
    public static void main( String[] args )
    {
        Function<Integer, Integer> multiply = null;
        System.out.println("Options:\n1) multiply 10 by 2\n2) multiply 10 by 4\nYour option: ");
        Scanner scanner = new Scanner(System.in);
        String option = scanner.nextLine();
        switch (option) {
            case "1": multiply = i -> i * 2; break;
            case "2": multiply = i -> i * 4; break;
            default: System.exit(0);
        }
        System.out.println(calculate(10, multiply));
    }
    public static Integer calculate(int a, Function<Integer, Integer> m) {
        return m.apply(a);
    }
}
```

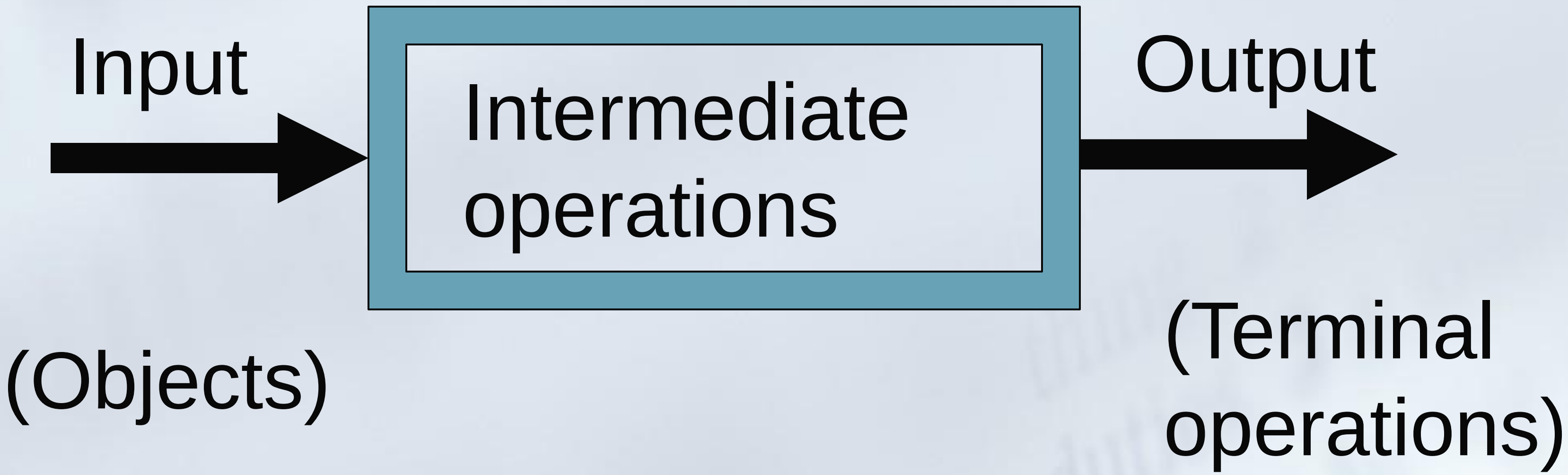
Stream API

New in Java 8 is the Stream API:

Interface Stream<T>

- Sequence of object - pipeline
- Perform operations on the elements in the stream
- Will only run if there's a terminating operation (lazy instantiation)
- Intermediate operations transforms input
- Once the stream is terminated you cannot perform any more operations on it.

Stream API (continued...)



Generating Streams

There are many ways to generate a stream:

- ```
  Stream<Integer> stream = Stream.empty()
  ```
  Empty stream

- ```
  Stream<Integer> stream = Stream.of(1, 2, 3);
  ```
  Finite stream

- ```
  Stream<Double> stream = Stream.generate(Math::random);
  ```
  Infinite stream

- ```
  Stream<Integer> stream = Stream.iterate(1, i -> i + 1);
  ```
  Generates an infinite stream of integers starting from 1

- ```
  List<String> l = Arrays.asList(“a”, “b”, “c”);
  Stream<String> = l.stream();
  ```
  ```

Terminal operations

The ultimate result of the stream. Sets the stream in motion. Example:

```
System.out.println(
                Stream.of(1, 2, 3)
                .count()
        );
Result: 3
```

On infinite streams terminal operations can have different effects. E.g. count() does not terminate in infinite streams:

```
System.out.println(
                Stream.generate(Math::random)
                .count()
        );
Result: doesn’t stop. Needs stopping of process
```

Common terminal operations

Method	What Happens for Infinite Streams	Return Value	Reduction
`allMatch()` `/anyMatch()` `/noneMatch()`	Sometimes terminates	`boolean`	No
`collect()`	Does not terminate	Varies	Yes
`count()`	Does not terminate	`long`	Yes
`findAny()` `/findFirst()`	Terminates	`Optional<T>`	No
`forEach()`	Does not terminate	`void`	No
`min()/max()`	Does not terminate	`Optional<T>`	Yes
`reduce()`	Does not terminate	Varies	Yes

Intermediate operations

These are operations on each element of the stream. Examples include: limit(), map(). Example:

```
Stream.of("A", "B", "C")
        .filter(s -> s.startsWith("B"))
        .forEach(System.out::println);


Stream.of("A", "BC", "DEF")
        .map(String::length)
        .forEach(System.out::println);
  }

System.out.println(Stream.of("A", "BC", "DEF")
        .map(String::length)
        .collect(Collectors.toList()));
```

Functional vs imperative

The functional programming style is more clear in its intent, whereas the traditional style is more verbose, and some would say “boiler plate”:

```
Stream.of("A", "B", "C")
               .filter(s -> s.startsWith("B"))
               .forEach(System.out::println);


List<String> string = Arrays.asList("A", "B", "C");
      for (String s : string) {
          if (s.startsWith("B")) {
              System.out.println(s);
          }
      }
```

Optional<T>

Sometimes it's not known if any operation will return a result. Then it's possible that the result is null, with the possibility of a NullPointerException.

Solution: wrap the result in an object, and determine what to do with it: Optional.

Example:

```
Stream<String> strings = Stream.of("Fender", "Gibson", "Ibanez");
Optional<String> optionalString = strings
        .filter(a -> a.startsWith("F"))
        .findAny();
System.out.println(optionalString);
Result: Optional[Fender]
```

Getting an optional

Optional is like a box. Once you have a result, you have to open the box, and get it out. But there's a danger:

```
Stream<String> guitars = Stream.of("Fender", "Gibson", "Ibanez");
String guitar = guitars
        .filter(a -> a.startsWith(“A"))
        .findAny()
        .get();
System.out.println(guitar);
Result:
Exception in thread "main" java.util.NoSuchElementException: No value
present
        at java.util.Optional.get(Optional.java:135)
        at nl.wimmelstein.Application.main(Application.java:15)
```

Optional... or else

It's possible to protect against these kind of exceptions in a number of ways:

```
Stream<String> guitars = Stream.of("Fender", "Gibson", "Ibanez");
String guitar = guitars
                        .filter(a -> a.startsWith("A"))
                        .findAny()
                        .orElse("No result");
System.out.println(guitar);
Result:
No result
```

Exercise

1. From the application you created last week, create an endpoint that will return a list of all items that fulfill a certain criteria, e.g. from demo last week: Return all guitars that are Fender

2. Using techniques from this lesson, create some new functionality of your own design. Expand the model if you have to.

Upload to git