Java fundamentals

Section 5 - Functional programming in Java

Topics

- Optional
- Functional programming
- Streams
- Functional interfaces

Optional

Optional

What is Optional?

- Introduced in Java 8
- Represents objects that can be null
- More on optionals here

Functional programming

Functional programming

What is functional programming?

- **Functional programming** is a programming paradigm where the program is constructed by building the functionality as if they were mathematical functions
- function is an expression that relates an **input** set to an **output** set.
- the output of a function depends only on its input.
- we can **compose** two or more functions together to get a new function.
- Starting with Java 8, there is support for functional programming in Java

Functional programming

Principles & Concepts

- Functions are **first-class** - functions are allowed to support all operations typically available to other entities, including:
  - Passing a function as a parameter to other functions
  - Returning a function as a result of a method
  - Assigning a function to a variable
- **Lambda** function = anonymous function

```
List<Employee> employees = new ArrayList<>();
//lambda expression
Collections.sort(employees, (employee1, employee2) -> {
    if (employee1.getFirstName().compareTo(employee2.getFirstName()) > 0) {
        return 1;
    } else if (employee1.getFirstName().compareTo(employee2.getFirstName()) < 0) {
        return -1;
    } else if (employee1.getSalary().compareTo(employee2.getSalary()) > 0) {
        return 1;
    } else if (employee1.getSalary().compareTo(employee2.getSalary()) < 0) {
        return -1;
    } else {
        return 0;
    }
});
```

Functional programming

Principles and concepts

- pure function should:
  - return a value based only on its arguments
  - have no **side effects**
- Side effects = anything apart the intended behaviour of the method (eg: updating a field of a class, or storing information into the database before returning the result

```
public Integer computeBudget(List<Employee> employees) {
    return employees.stream().map(Employee::getSalary).mapToInt(Integer::intValue).sum();
}
```

Functional programming

Principles and concepts

- **Immutability** = entities can’t be modified after they have been instantiated
- **Referential transparency** = an expression referentially transparent if replacing it with its corresponding value has no impact on the program's behaviour (we need pure functions and immutability to achieve this)

- More about functional programming here
- More about lambdas here

Streams

Streams

What is a Stream?

- Package: ***java.util.stream***
- Stream = sequence of elements, flux of elements
- Class: ***Stream<T>***
- Streams can be created from collections
- Stream operations:
  - **Intermediate operations**: return a new Stream
    - Intermediate operations can be chained
  - **Terminal operations**: return a result

Streams

Intermediate operations

- map() : converts each element from the Stream to a new element, by applying a function
- filter() : filters the elements from the stream using a Predicate function (condition)
- sorted() : returns a sorted stream
- distinct() : returns only the unique elements from the stream
- flatMap() : converts a Stream of collections into a Stream

Streams

Terminal operations

- foreach() : iterates over a stream and applies a function to each element
- collect() : collects the stream into a collection
- match() : checks if the elements from the stream match with a Predicate; returns boolean
- count() : counts the number of elements from a stream
- reduce() : applies a function in order to return the stream to only one element; returns Optional
- min(), max(), average(

- More about streams here and here

Functional interfaces

Functional interface

- Functional interface - interface that contains exactly one abstract method
- They can be implemented using lambda expressions
- java.util.function - predefined functional interfaces
- We can define custom functional interfaces
  - We should annotate them with @FunctionalInterface; if we do, the compiler will throw an error in case there is more than 1 abstract method defined
- More here

Functional interfaces

- **Function**
  - One parameter
  - One result
  - Abstract method: apply(Object)
- **BiFunction**
  - Two parameters
  - One result
  - Abstract method : apply(Object, Object);
  - IntFunction, DoubleFunction, IntToDoubleFunction, IntToLongFunction, DoubleToIntFunction, DoubleToLongFunction, LongToDoubleFunction, and LongToIntFunction.

Functional interfaces

- **Predicate**
  - One parameter
  - Result: boolean
  - Abstract method: test(Object)
- **BiPredicate**:
  - Two parameters, result: boolean
- **Supplier**:
  - No parameter
  - One result
  - Abstract method: get()
  - IntSupplier, DoubleSupplier, BooleanSupplier, LongSupplier

Functional interfaces

- **Consumer**:
  - One parameter
  - No result
  - Abstract method: accept(Object)
  - IntConsumer, LongConsumer, DoubleConsumer, BiConsumer, ObjtIntConsumer, ObjLongConsumer, ObjDoubleconsumer
- **UnaryOperator**
  - One parameter, one result (same type)
  - Abstract method: apply(Object)
  - IntUnaryOperator, DoubleUnaryOperator, LongUnaryOperator
- **BinaryOperator**
  - Two parameters, one result (same type)
  - Abstract method apply(Object, Object)
  - IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator