# FUNCTIONAL PROGRAMMING

# CONTENT

- Collections streams and filters
- Iterate using forEach
- Streams API
- Built-in functional interfaces
- Optionals

# VARIABLES IN LAMBDAS

- Lambda expressions have the same rules for accessing variables as inner classes. They can access:
  - Final and effectively final local variables
  - Static variables
  - Instance variables
  - Method parameters

- Effectively final variable: not declared as final, but value never changes after initialization

# BUILT-IN FUNCTIONAL INTERFACES

- Functional interface is an interface with exactly one abstract method
- Built-in interfaces are in the java.util.function package
- Many interfaces, for generic interfaces, but also for primitives
- Used a lot for streams

# SUPPLIER

```
@FunctionalInterface public interface Supplier<T>{
        public T get();
}
```

Used for supplying values without needing an input parameter

# EXAMPLE SUPPLIER

```
Supplier <Animal> supplier = () - > new Animal(“Jimmy”, “Spider”);
```

```
Animal a = supplier.get();

System.out.println(“Animal:" + a.getName() + “ is a " + a.getType());
```

# EXERCISE SUPPLIER

Print 5 random values using a supplier

Bonus: write it twice, once with a lambda and another time with a method reference

# CONSUMER AND BICONSUMER

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

@FunctionalInterface public interface BiConsumer<T, U> {
        void accept(T t, U u);
}
```

Used for doing something with a parameter without returning anything.

# EXAMPLE CONSUMER AND BICONSUMER

```
Consumer<String> c = x -> System.out.println(x);

c.accept("Hi there");
```

# EXERCISE CONSUMER AND BICONSUMER

Use a Consumer to double the integers in a list

# LOOPING THROUGH COLLECTION WITH LAMBDA (CHAPTER 3)

collectionName.forEach(Consumer c)

The interface Consumer takes a single parameter and doesn't return anything

Exercise: print the names of all cats in a collection of animals

# PREDICATE AND BIPREDICATE

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

@FunctionalInterface public interface BiPredicate<T, U> {
        boolean test(T t, U u);
}
```

Predicate is used to test for a certain condition, often for filtering and matching

# EXAMPLE PREDICATE AND BIPREDICATE

```
Predicate<String> p = str -> str.isEmpty();

p.test("Hi"); //false

p.test(""); //true
```

# EXERCISE PREDICATE AND BIPREDICATE

Write a BiPredicate to test whether two animals have the same name

Bonus: also write a bipredicate that checks whether two animals have the same color

Bonus bonus: write a new bipredicate that uses that two others to check for the same color and same name, and another that checks for the same name, but different color. Avoid duplicate code by using default methods of the predicate interfaces

# REMOVING CONDITIONALLY (CHAPTER 3)

Boolean removeIf(Predicate<? super E> filter)

Can be used to remove items from a collection based on a certain condition

Exercise: remove cats from your collection based on  a certain condition (e.g. names starting with a C or the color being white)

# FUNCTION AND BIFUNCTION

```
@FunctionalInterface public interface Function<T, R> {
        R apply(T t);
}

@FunctionalInterface public interface BiFunction<T, U, R> {
        R apply(T t, U u);
}
```

Used to change the value of the input parameter to something else, possibly something of a different type

# EXAMPLE FUNCTION AND BIFUNCTION

```
BiFunction<String, String, String> bf = (x, y) -> {
    return x + y;
};

System.out.println(bf.apply(“Hi ", “there"));
```

# EXERCISE FUNCTION AND BIFUNCTION

Write a function that takes a string and a substring, and that returns the number of times the substring occurs in the string

# JAVA 8 MAP API (CHAPTER 3)

- merge()
  - mapName.merge(obj1, obj2, biFunctionImpl);
  - BiFunction interface takes two parameters and returns a value
- computeIfPresent() -> not on OCP
  - BiFunction<T, U, V> mapper = some implementation
  - mapName.computeIfPresent(value, mapper)
  - Removes key if result of mapper is null
- computeIfAbsent() -> not on OCP
  - Function<T, U> mapper = some implementation
  - mapName.computeIfAbsent(key, mapper)

# UNARYOPERATOR AND BINARYOPERATOR

```
@FunctionalInterface public interface UnaryOperator<T> extends Function<T, T> {
}
```

```
@FunctionalInterface public interface BinaryOperator<T> extends BiFunction<T, T, T> {
}
```

The method is in the function and bifunction interface. The difference with function and bifunction is that input and output must be of the same type. And for binaryoperator both input parameters are of the same type.

Used for changing the value of the input to something of the same type.

# EXAMPLE UNARYOPERATOR AND BINARYOPERATOR

```
UnaryOperator<String> uo  = (x)-> x.toUpperCase();

System.out.println(uo.apply("Hi there"));
```

# EXERCISE UNARYOPERATOR AND BINARYOPERATOR

Write a unaryoperator that takes the name of a document, if it doesn't have the suffix .doc or .docx it should add the suffix .docx. It should return the name of the document with the suffix.

# UPDATING ALL ELEMENTS IN A LIST (CHAPTER 3)

void replaceAll(UnaryOperator<E> o)

Used for replacing all elements with another value

The interface UnaryOperator takes a parameter of a certain type and returns a parameter of the same type

Example: listName.replaceAll(x -> x+1)

Exercise: replace all cat names with your favorite cat name

# EXERCISE: WHICH BUILT-IN FUNCTIONAL INTERFACE SHOULD YOU USE?

1. Takes an Integer and returns an Integer
2. Returns a boolean and takes a String
3. Return an Animal object without taking any input
4. Print the Animal name and return nothing
5. …. <String, Integer> si = (str, int1) -> true;
6. …. <String, Integer, Boolean> sib = (str, int1) -> true;
7. …. <String> s = s -> System.out.println(s);
8. … <Animal> a = Animal::new;

# OPTIONAL

- Special data type that can be used when a value might not be there
- Java's way of expressing "not applicable"
- Created using a factory
- It is like a box for another data type, and that box might be empty

# OPTIONAL EXAMPLE

```
Optional<Animal> o = Optional.empty(); //there's no animal in here

Animal a = new Animal();

a.setName("Roxy");

Optional<Animal>o2 = Optional.of(a);
```

# OPTIONAL EXERCISE

- Create a string name
- Make a method that takes that name as input
- If name is empty, it will return an empty optional (instead of null)
- Else it will return an optional with the value of the input

# OPTIONAL METHODS

| Method name | Optional empty | Optional has value |
|---|---|---|
| get() | Throws exceptions | Returns value |
| ifPresent(Consumer c) | Nothing | Calls consumer with the value as parameter |
| isPresent() | Returns false | Returns true |
| orElse(T t) | Returns whatever is specified | Returns value |
| orElseGet(Supplier s) | Returns result of calling supplier | Returns value |
| orElseThrow(Supplier s) | Throws exception created by supplier | Returns value |

# OPTIONAL METHODS EXAMPLE

```
Optional<Integer> opt = Optional.ofNullable(getScore());

System.out.println(opt.orElseGet(100));

System.out.println(opt.orElseThrow(()->new IllegalStateException()));
```

# OPTIONAL METHODS EXERCISE

- Rewrite the following code with optional:

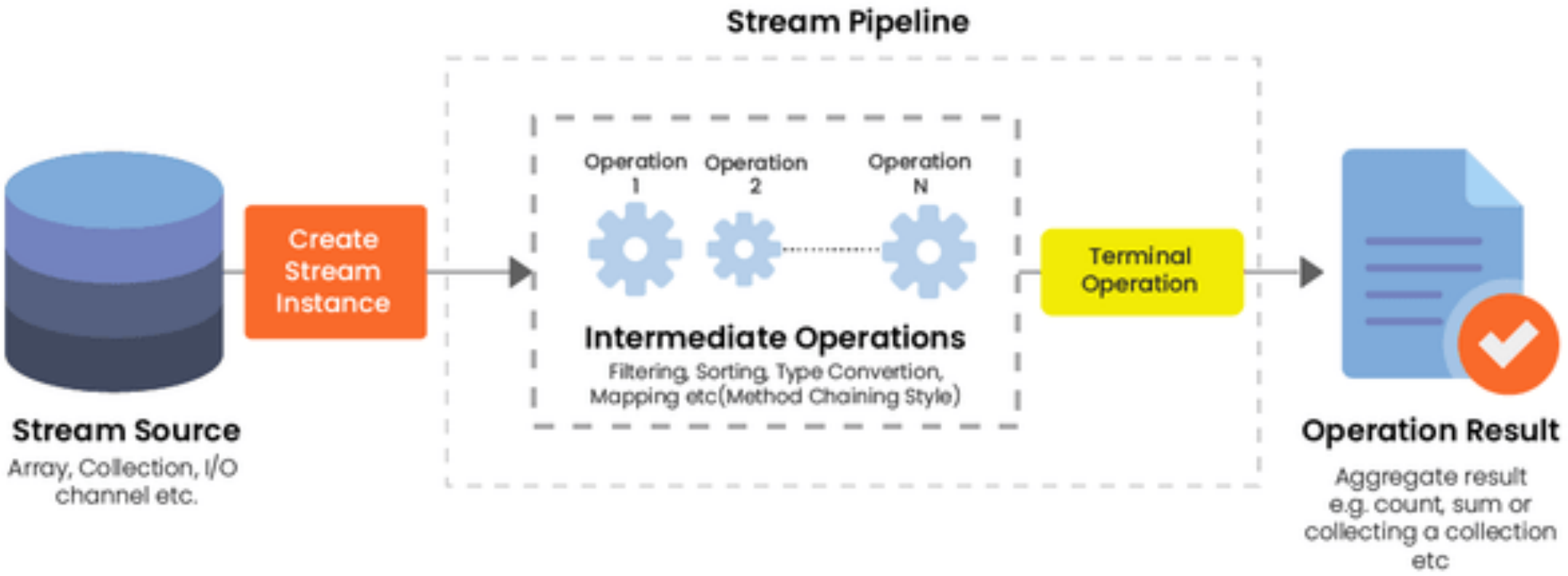
```
Car c = new Car(“MaaikesCar", new Engine(hp, fuel, etc);
Engine e = c.getEngine();


if(c != null){
 System.out.println(c);
}
```

# STREAMS

- Flow of data that is being created when needed
- Finite streams contain limited amount of data, infinite streams have an unlimited amount of data (like random number generating)
- Stream consists of:
  - Source: where the data of the stream comes from
  - Intermediate operations: transfers stream, only evaluated when the terminal operation runs as well
  - Terminal operation: produces the result, nothing can be changed to the stream after a terminal operation

# Java Streams



# STREAM SOURCES

- Finite stream:

```
Stream<Integer> integerStream = Stream.of(1, 2, 3, 4, 5);
Stream<String> valuesFromList = listName.stream();
```

- Infinite stream:

```
Stream<Double> randomValues = Stream.generate(Math::random);
Stream<Integer> evenNrs = Stream.iterate(2, n -> n + 2);
```

.parallelStream() >> creates a stream that can be processed parallel

# EXERCISE: EMPTY STREAMS

- You can create an empty stream like this:

```
Stream<String> empty = Stream.empty();
```

Why would you do this?

# STREAM TERMINAL OPERATIONS

- Only stream source + terminal operation is a valid stream
- Terminal operations end the stream
- Reductions are a special type: they “reduce” the entire stream to a single object

# STREAM TERMINAL OPERATIONS

| Method | Infine stream terminated? | Return value | Reduction |
|---|---|---|---|
| allMatch(), anyMatch(), noneMatch() | Sometimes | boolean | no |
| collect() | No | Varies | Yes |
| count() | No | long | Yes |
| findAny(), findFirst() | Yes | Optional<T> | No |
| forEach() | No | void | No |
| min(), max() | No | Optional<T> | Yes |
| reduce() | No | Varies | yes |

# EXERCISE: STREAM TERMINAL OPERATIONS

Create an infinite stream of numbers and a finite stream of Strings and test each by using the following functions to do:

allMatch() >> write to find all with the third letter of the string being an F / being higher than 6

anyMatch() >> find any number between 77 and 100 and any string starting with a T

noneMatch() >> see if none matcch a string longer than 25, see if none match 5

collect() >> collect all the string in a stringbuilder and sum all the even integers

count() >> count the stream and print the result

findAny() >> see what happens for empty and when there are values

findFirst() >> idem

forEach() >> see what happens for printing on the finite and infinite stream

min() >> find the shortest String

max()>> find the longest String

reduce() >> reduce the string stream in a long string consisting of the first letters of all string

# STREAM TERMINAL OPERATIONS

- Only stream source + terminal operation is a valid stream
- Terminal operations end the stream
- Reductions are a special type: they “reduce” the entire stream to a single object

# STREAM TERMINAL OPERATIONS

| Method | Infine stream terminated? | Return value | Reduction |
|---|---|---|---|
| allMatch(), anyMatch(), noneMatch() | Sometimes | boolean | No |
| collect() | No | Varies | Yes |
| count() | No | long | Yes |
| findAny(), findFirst() | Yes | Optional<T> | No |
| forEach() | No | void | No |
| min(), max() | No | Optional<T> | Yes |
| reduce() | No | Varies | yes |

# EXERCISE: STREAM TERMINAL OPERATIONS

Create an infinite stream of numbers and a finite stream of Strings and test each by using the following functions to do:

allMatch() >> write to find if all have the third letter of the string being an F / are higher than 6

anyMatch() >> find any number between 77 and 100 and any string starting with a T

noneMatch() >> see if none matcch a string longer than 25, see if none match 5

collect() >> collect all the string in a stringbuilder and sum all the even integers

count() >> count the stream and print the result

findAny() >> see what happens for empty and when there are values

findFirst() >> idem

forEach() >> see what happens for printing on the finite and infinite stream

min() >> find the shortest String

max()>> find the longest String

reduce() >> reduce the string stream in a long string consisting of the first letters of all string

Bonus: why empty Stream option?

# STREAM INTERMEDIATE OPERATIONS

- Intermediate operation return a stream
- They are not executed until a terminal operation is called upon the stream

# STREAM INTERMEDIATE OPERATIONS

- Stream<T> filter(Predicate<? super T> predicate)

Returns stream with only values that were true in predicate

- Stream<T> distinct()

Returns stream without double values

- Stream<T> limit(int maxSize)

Limit the stream to a specific number

- Stream<T> skip(int n)

Skips the first n elements of the stream and returns a stream without these first n elements

- Stream<T> sorted() and Stream<T> sorted(Comparator<? super T> comparator)

Returns elements in natural order, unless specific Comparator is specified

# EXERCISE: STREAM INTERMEDIATE OPERATIONS

Create an infinite stream of integers and a finite stream of Strings and test each by using the following functions to do, before printing with a foreach

Filter() >> filter the strings on length longer than 5 and numbers on even numbers

Distinct() >> remove duplicates in both streams

Limit() >> limit the streams to 5

Sorted and skip() >> sort the stream and skip the first few elements

# STREAM INTERMEDIATE OPERATIONS

- <R> Stream<R> map(Function<? super T, ? extends R> mapper)

Returns a stream with values as changed by the function

- <R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)

Returns a stream of the type that the function returns, removes all list and places objects of underlying lists at the same level

- Stream<T> peek(Consumer<? super T> action)

Doesn't change the stream, allows to print/store values

# EXERCISE: STREAM INTERMEDIATE OPERATIONS

Create a stream with lists of lists and a stream with Strings

Map() >> reverse all the strings

flatMap() >> get all elements on the same level

Peek >> print inbetween by using peek and end with a count()

# STREAM PIPELINE

- Consists of:
  - Stream source
  - Optional: intermediate operataions
  - Terminal operation
- Method chaining, the methods of a stream can be chained and show a very readable way of operating on streams of data

# STREAM EXERCISES

- Create a stream from a list with cats
- Remove the cats with long hair (or short, whatever you prefer)
- Remove duplicates
- Sort the cats on first name, reversed alphabetical order
- Create a list of these cats

Bonus

- Create a stream of sequential odd numbers starting at 1 (1,3,5,etc)
- Only have 100 odd numbers in the stream
- Remove all the non-prime numbers
- Remove all numbers above 100
- Count the prime numbers under a 100

# STREAMS AND PRIMITIVES

- Instead of wrapper classes, streams can also work with primitives
- Primitive streams have special methods that are helpful for the working with primitives (.average(), .range(), .summaryStatistics)

- DoubleStream
- IntStream
- LongStream

# CREATE PRIMITIVE STREAMS

```
DoubleStream ds = DoubleStream.of(1.0, 1.1, 1.2);

DoubleStream random = DoubleStream.generate(Math::random);
```

# EXERCISE

Create a stream of doubles and calculate the sum

Create  stream of integers and find the max value

Create a stream of long and find the min value

Create a stream of char and find max value

Bonus: which ones don't work on infinite streams?

# MAPPING METHODS BETWEEN DIFFERENT STREAMS

| Source class | To Stream | To DoubleStream | To IntStream | To LongStream |
|---|---|---|---|---|
| Stream | map | mapToDouble | mapToInt | mapToLong |
| DoubleStream | mapToObj | map | mapToInt | mapToLong |
| IntStream | mapToObj | mapToDouble | map | mapToLong |
| LongStream | mapToObj | mapToDouble | mapToInt | map |

Parameters for mapping methods:

| Source class | To Stream | To DoubleStream | To IntStream | To LongStream |
|---|---|---|---|---|
| Stream | Function | ToDoubleFunction | ToIntFunction | ToLongFunction |
| DoubleStream | DoubleFunction | DoubleUnaryOperator | DoubleToIntFunction | DoubleToLongFunction |
| IntStream | IntFunction | IntToDoubleFunction | IntUnaryOperator | IntToLongFunction |
| LongStream | LongFunction | LongToDoubleFunction | LongToIntFunction | LongUnaryOperator |

# EXERCISE: STREAMS AND PRIMITIVES

- Make an infinite stream of doubles
- Calculate the average
- Print the value

- Make an infinite streams of integers without making an intstream
- Create an IntStream from this stream
- Limit the stream to 100 values
- Find and get the max value

- What happens when you limit the stream to 0?

Bonus:

Find two ways to create a Stream<Integer> from an intstream.

# BONUS EXERCISE: INTSTREAM TO STREAM<INTEGER>

- Find two ways to create a Stream<Integer>  from an intstream.

# OPTIONALS AND PRIMITIVE STREAMS

| | OptionalDouble | OptionalInt | OptionalLong |
|---|---|---|---|
| Get primitive value | getAsDouble() | getAsInt() | getAsLong() |
| orElseGet's parameter | DoubleSupplier | IntSupplier | LongSupplier |
| Return type of max() | OptionalDouble | OptionalInt | OptionalLong |
| Return type of sum() | double | int | long |
| Return type of average() | OptionalDouble | OptionalDouble | OptionalDouble |

# PRIMITIVE STREAMS USE OWN BUILT-IN FUNCTIONAL INTERFACES

| Functional interfaces | Parameters | Return type | Single methodS |
|---|---|---|---|
| Double/Int/LongSupplier | 0 | double, int, long | getAsDouble/-Int/-Long |
| Double/Int/LongConsumer | 1 (double, int, long) | void | Accept |
| Double/Int/LongPredicate | 1 (double, int, long) | boolean | Test |
| Double/Int/LongFunction | 1 (double, int, long) | R | Apply |
| Double/Int/LongUnaryOperator | 1 (double, int, long) | double, int, long | applyAsDouble/-Int/-Long |
| Double/Int/LongBinaryOperator | 2 (2x double, 2x int, 2x long) | double, int, long | applyAsDouble/-Int/-Long |

# COLLECTING RESULTS

- Basic collectors
- Into maps
- Grouping, partitioning and mapping
  - groupingBy()
  - Collectors.toSet/.toList/

# ADVANCED STREAM PIPELINE CONCEPTS

- Chaining optionals
- Collecting results

# EXERCISE

- Create a new map to store first names and favorite art form.
- Use merge to add values of art form for the first names
- Use ComputeIfPresent to add ice skating as default for names starting with an M or S or J
- Use ComputeIfAbsent to add a new pair
- Use ComputeIfAbsent on an existing pair