# JDK8 Functional Programming
# Streams API

Michał Rudnik

# Agenda

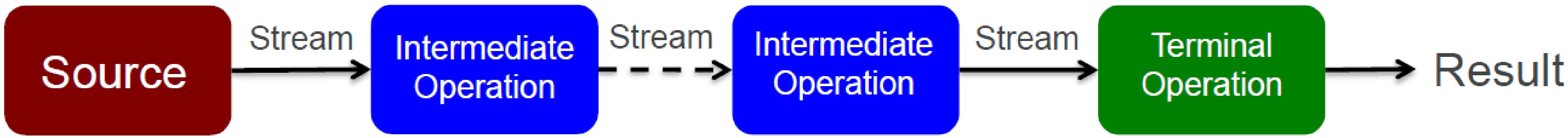
- Elements of a `Stream`
- Streams of objects and primitive types
- `Stream` sources in JDK 8
- `Stream` interface: intermediate operations
- `Stream` interface: terminal operations
- `Optional` class

# Stream overview

- Abstraction for specifying aggregate computations
- Allows processing data in a declarative way
- Represents a sequence of elements from a source, which supports aggregate operations
- Computes elements on demand
- Not a data structure, does not store the elements
- Can be infinite

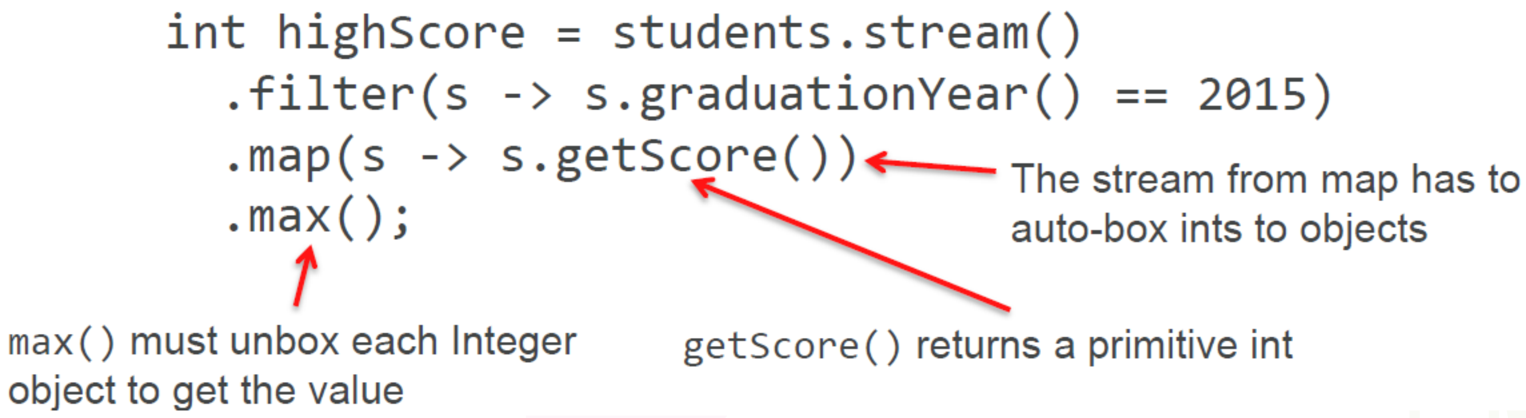
# Elements of a `Stream`

- A stream pipeline consists of three types of things:
  - source
  - zero or more intermediate operations
  - terminal operation (producing a result or a side-effect)



# Streams of objects and primitive types

- By default, a stream produces elements that are objects
- Sometimes this is not the best solution



```
int highScore = students.stream()
  .filter(s -> s.graduationYear() == 2015)
  .map(s -> s.getScore())
  .max();
```

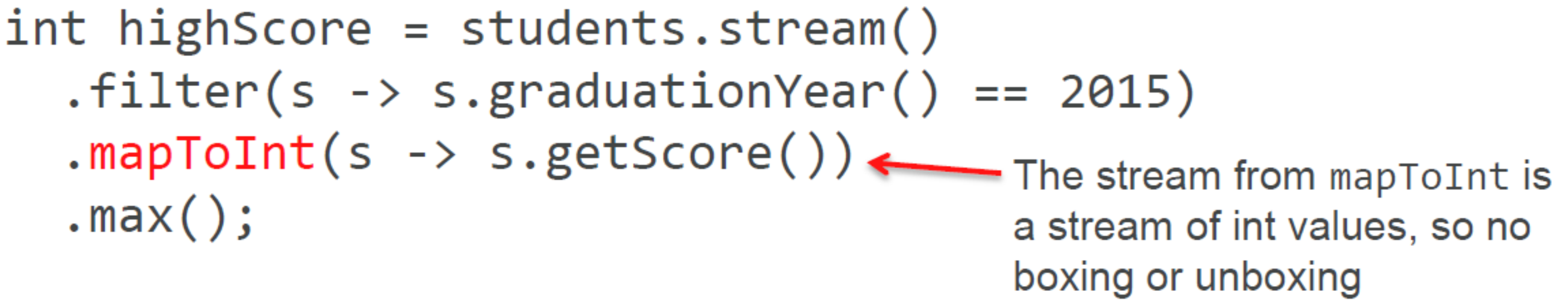
The stream from map has to auto-box ints to objects

max() must unbox each Integer object to get the value

getScore() returns a primitive int

# Streams of objects and primitive types

- Three primitive stream types:
  - `IntStream, DoubleStream, LongStream`
- Avoiding a lot of unnecessary object creation
- Improving stream efficiency
- Use `mapToInt(), mapToDouble(), mapToLong()`



```
int highScore = students.stream()
  .filter(s -> s.graduationYear() == 2015)
  .mapToInt(s -> s.getScore())
  .max();
```

The stream from `mapToInt` is a stream of int values, so no boxing or unboxing

# Stream sources in JDK8

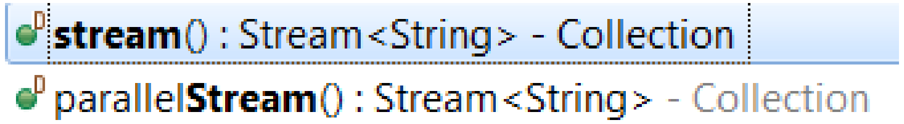
- There are 95 methods in 23 classes that return a `Stream`
  - many of them are just intermediate operations in the `Stream` interface

- 71 methods in 15 classes can be used as practical `Stream` sources

# Collection interface

- **stream()**
  - provides a sequential stream of elements in the collection

- **parallelStream()**
  - provides a parallel stream of elements
  - uses the fork-join framework for implementation

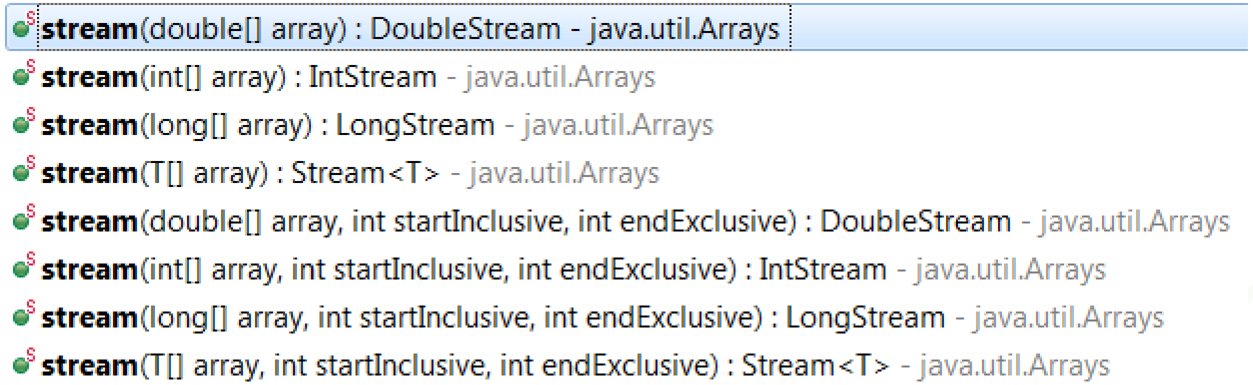


stream() : Stream<String> - Collection

parallelStream() : Stream<String> - Collection

# `Arrays` class

- `stream()`
  - an array is not a collection in sense of Java Collections API
  - still you can create a stream out of it
  - provides a sequential stream
  - static methods of the `Arrays` class
  - overloaded methods for different types



**stream**(double[] array) : DoubleStream - java.util.Arrays

**stream**(int[] array) : IntStream - java.util.Arrays

**stream**(long[] array) : LongStream - java.util.Arrays

**stream**(T[] array) : Stream<T> - java.util.Arrays

**stream**(double[] array, int startInclusive, int endExclusive) : DoubleStream - java.util.Arrays

**stream**(int[] array, int startInclusive, int endExclusive) : IntStream - java.util.Arrays

**stream**(long[] array, int startInclusive, int endExclusive) : LongStream - java.util.Arrays

**stream**(T[] array, int startInclusive, int endExclusive) : Stream<T> - java.util.Arrays

# Files class

- **find()**
  - stream of File references that match a given BiPredicate

- **list()**
  - stream of entries from a given directory

- **lines()**
  - stream of strings that are the lines read from the given file

# Random Numbers

- Three random related classes
  - `Random`, `ThreadLocalRandom`, `SplittableRandom`

- Methods to produce finite of infinite streams of random numbers
  - `ints()`, `doubles()`, `longs()`
  - four versions of each
    - finite of infinite
    - with and without seed

# Other classes and methods

- `JarFile/ZipFile: stream()`
  - returns a `File` stream of the contents of the compressed archive
- `BufferedReader: lines()`
  - returns a stream of strings that are the lines read from the input
- `Pattern: splitAsStream()`
  - returns a stream of strings of matches of a pattern
  - like `split()`, but returns a stream instead of an array

# `Stream` static methods

- `concat(Stream, Stream)`
  - concatenates two specified streams

- `empty()`
  - returns an empty stream

- `of(T... values)`
  - stream that consists of the specified values

- `range(int, int)`, `rangeClosed(int, int)`
  - stream from a start to an end value (exclusive or inclusive)

# Intermediate operations – filtering and mapping

- Powerful range of intermediate operations allow streams to be manipulated as required

- **`distinct()`**
  - returns a stream with no duplicate elements

- **`filter(Predicate p)`**
  - returns a stream with only those elements evaluate as `true` for the `Predicate`

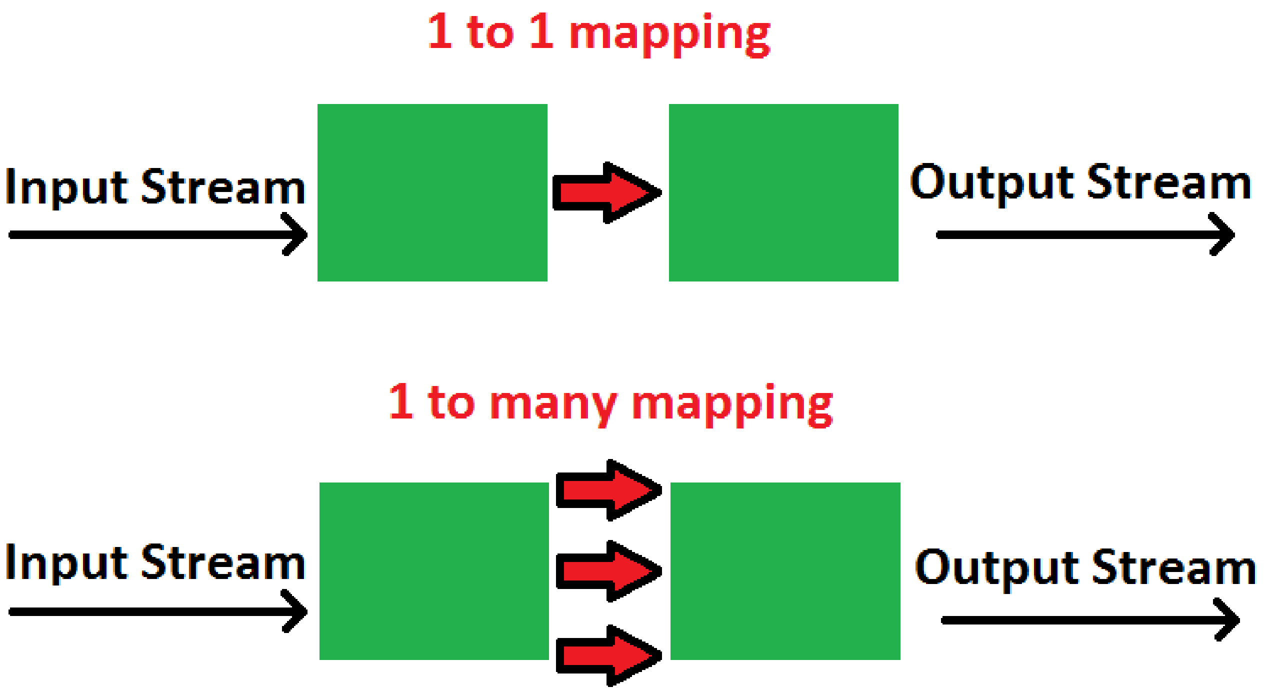
- **`map(Function f)`**
  - returns a stream where the given `Function` is applied to each element of the input stream

- **`mapToInt(), mapToDouble(), mapToLong()`**
  - like `map()`, but producing streams of primitives rather than objects

# Map and FlatMap

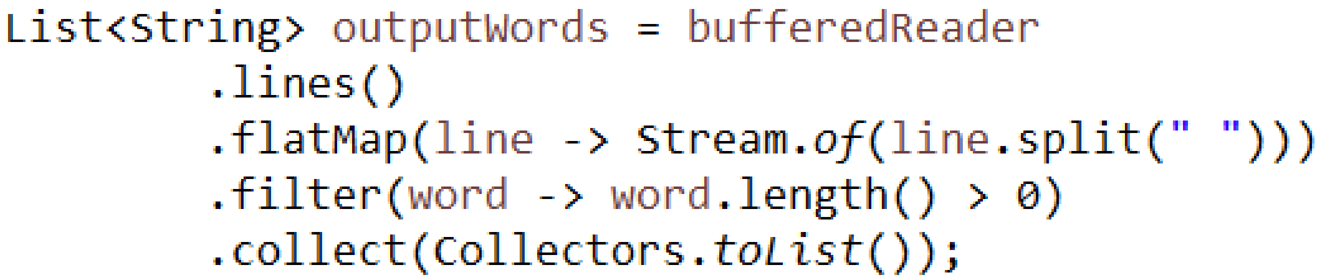
Map

FlatMap



# FlatMap example

- BufferedReader returns Stream of Strings (file lines)
- Every line is mapped to Stream of Strings (line splitted by spaces)
- Using map() would create Stream of Streams
- All output Streams in flatMap() are concatenated into one output Stream

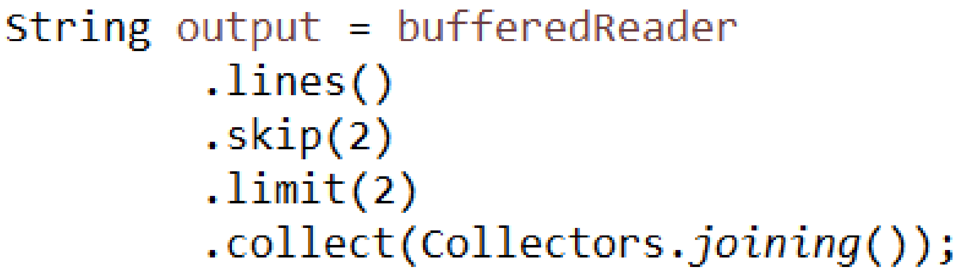


```
List<String> outputWords = bufferedReader
        .lines()
        .flatMap(line -> Stream.of(line.split(" ")))
        .filter(word -> word.length() > 0)
        .collect(Collectors.toList());
```

# Restricting the size of a `Stream`

- **`skip(long n)`**
  - returns a stream that skips the first *n* elements of the output stream

- **`limit(long n)`**
  - returns a stream that only contains the first *n* elements of the input stream



```
String output = bufferedReader
        .lines()
        .skip(2)
        .limit(2)
        .collect(Collectors.joining());
```

# Sorting and unordering

- **`sorted(Comparator c)`**
  - returns a stream that is sorted with the order determined by the `Comparator`
  - `sorted()` with no arguments sorts by natural order

- **`unordered()`**
  - returns a stream that is unordered (used internally)
  - does not change the order of stream elements, just the stream characteristics
  - can improve the efficiency of operations like `distinct()` and `groupingBy()`

# Observing Stream elements

- **peek(Consumer c)**
    - returns an output stream that is identical to the input stream
    - each element is passed to the accept() method of the Consumer
    - Consumer must not modify the elements of the stream
    - useful for debugging

# Terminal operations

- Terminates the pipeline of operations on the stream
- Only at this point any processing is executed
  - this allows for optimisation of the pipeline
    - lazy evaluation
    - fused operations
    - elimination of redundant operations
    - parallel execution
- Generates an explicit result or a side effect

# Matching elements

- **findFirst(Predicate p)**
  - the first element that matches using given Predicate
- **firstAny(Predicate p)**
  - the same as findFirst(), but for a parallel stream
- **boolean allMatch(Predicate p)**
  - whether all the elements of the stream match using the Predicate
- **boolean anyMatch(Predicate p)**
  - whether any of the elements of the stream match using the Predicate
- **boolean noneMatch(Predicate p)**
  - whether no elements match using the Predicate

# Collecting results

- **`collect(Collector c)`**
  - performs a mutable reduction on the stream
  - many existing collectors in `Collectors` class
    - `Collectors.joining()`
    - `Collectors.toList()`, `Collectors.toSet()`, `Collectors.toCollection()`
    - `Collectors.toMap()`
    - `Collectors.groupingBy(), Collectors.partitioningBy()`

- **`toArray()`**
  - returns an array containing the elements of the stream

# Numerical results

- **`count()`**
  - returns how many elements are in the stream

- **`max(Comparator c)`**
  - maximum value element of the stream using `Comparator`
  - returns an `Optional`, since the stream may be empty

- **`min(Comparator c)`**
  - minimum value element of the stream using `Comparator`
  - returns an `Optional`, since the stream may be empty

# Numerical results - primitive type streams

- **average()**
  - return the arithmetic mean of the stream
  - returns an OptionalInt, OptionalLong, etc. as the stream may be empty
- **sum()**
  - returns the sum of the stream elements
- **max()**
  - maximum value element of the stream
  - Comparator not needed, returns an OptionalInt, OptionalLong etc.
- **min()**
  - minimum value element of the stream
  - Comparator not needed, returns an OptionalInt, OptionalLong etc.

# Iteration

- `forEach(Consumer c)`
  - performs an action for each element of the stream

- `forEachOrdered(Consumer c)`
  - like `forEach`, but ensures that the order of the elements (if one exists) is respected when used for a parallel stream

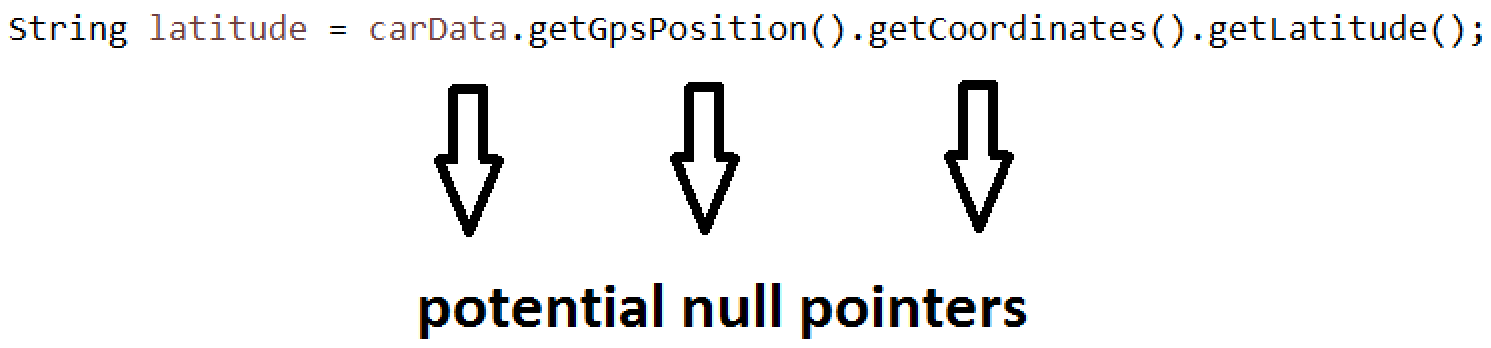
# Reduction

- Creating a single result from multiple input elements

- **`reduce(BinaryOperator accumulator)`**
  - performs a reduction on the stream using `BinaryOperator`
  - accumulator takes a partial result and the next element, and returns a new partial result
  - returns an `Optional`
  - also one version that takes an initial value (does not return an `Optional`)

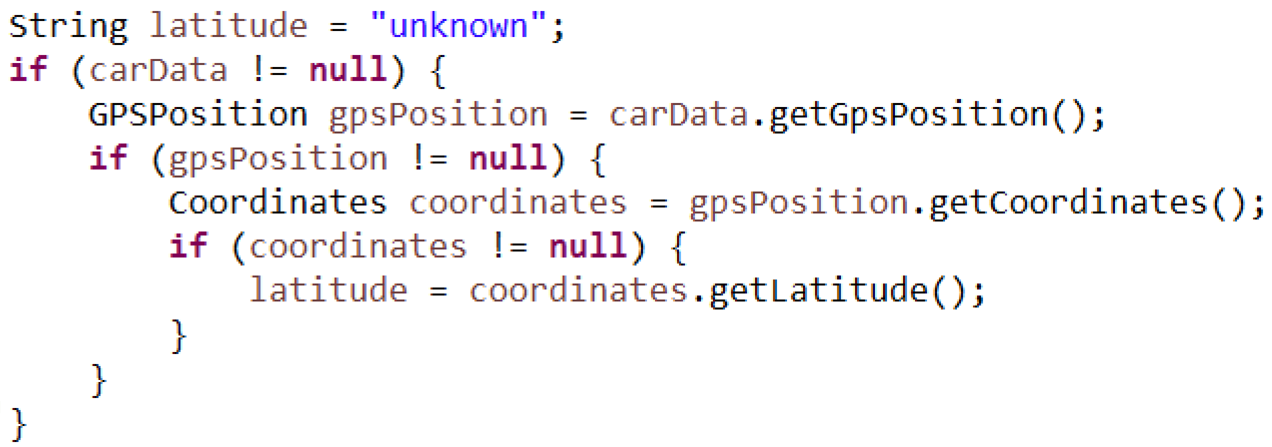
# Optional class

- Certain situations in Java return a result which is a null
- Problem with avoiding NullPointerException



```
String latitude = carData.getGpsPosition().getCoordinates().getLatitude();
```

potential null pointers



```
String latitude = "unknown";
if (carData != null) {
    GPSPosition gpsPosition = carData.getGpsPosition();
    if (gpsPosition != null) {
        Coordinates coordinates = gpsPosition.getCoordinates();
        if (coordinates != null) {
            latitude = coordinates.getLatitude();
        }
    }
}
```

# `Optional` class

- Helping to eliminate the `NullPointerException`
- Terminal operations like `min()` and `max()` may not return a direct result
  - suppose the input stream is empty

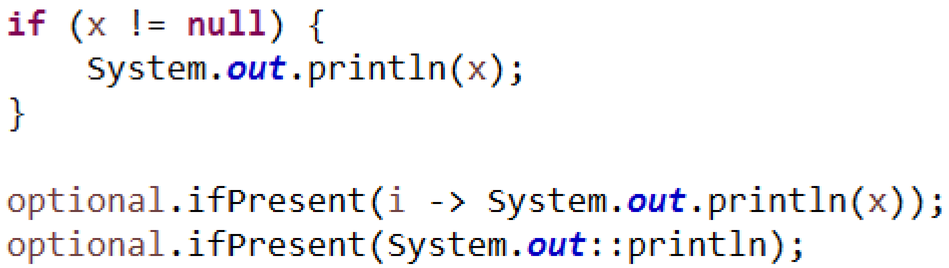
- **`Optional<T>`**
  - container for an object reference (`null` or real object)
  - think of it like a stream of 0 or 1 elements
  - guaranteed that `Optional` reference will not be `null`

# Optional class methods

- **static Optional<T> empty()**
  - returns an empty Optional instance
- **static Optional<T> of(T value)**
  - returns an Optional with the specified present non-null value
- **static Optional<T> ofNullable(T value)**
  - returns an Optional describing the specified value, if non-null
  - otherwise returns an empty Optional
- **T get()**
  - if the value is present in this Optional, returns the value
  - otherwise throws NoSuchElementException
- **T orElse(T other), orElseGet, orElseThrow**
  - return the value if present, otherwise return other or throw exception
- **boolean isPresent()**
  - return true if there is a value present, otherwise false

# Optional.ifPresent(Consumer c)

- If a value is present, invoke the specified Consumer with the value, otherwise do nothing

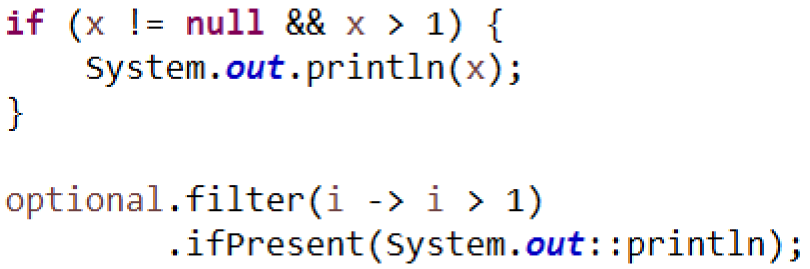


```
if (x != null) {
    System.out.println(x);
}

optional.ifPresent(i -> System.out.println(x));
optional.ifPresent(System.out::println);
```

# Optional.filter(Predicate p)

- If a value is present, and the value matches the given predicate, returns an Optional describing the value, otherwise returns an empty Optional

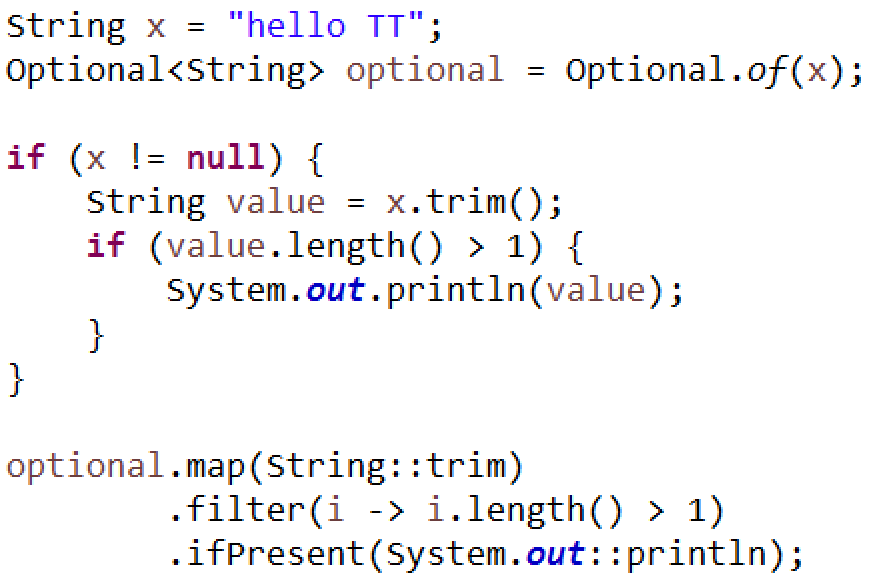


```
if (x != null && x > 1) {
    System.out.println(x);
}

optional.filter(i -> i > 1)
        .ifPresent(System.out::println);
```

# Optional.map(Function f)

- If a value is present, apply the provided mapping function to it, and if the result is non-null, return an `Optional` describing the result



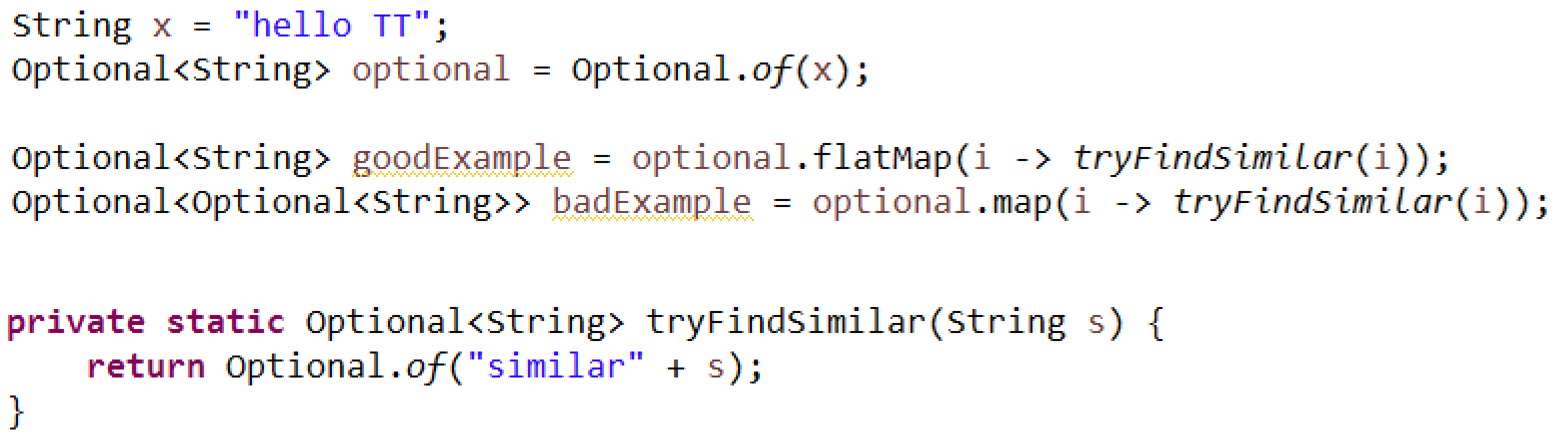
```
String x = "hello TT";
Optional<String> optional = Optional.of(x);

if (x != null) {
    String value = x.trim();
    if (value.length() > 1) {
        System.out.println(value);
    }
}

optional.map(String::trim)
        .filter(i -> i.length() > 1)
        .ifPresent(System.out::println);
```

# Optional.flatMap(Function f)

- Used when we want to apply map to something that already returns an Optional



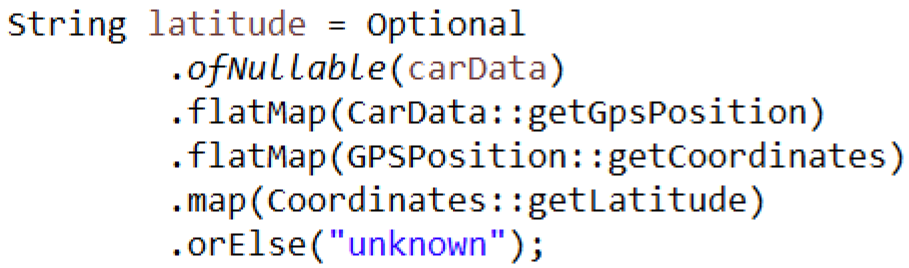
```
String x = "hello TT";
Optional<String> optional = Optional.of(x);

Optional<String> goodExample = optional.flatMap(i -> tryFindSimilar(i));
Optional<Optional<String>> badExample = optional.map(i -> tryFindSimilar(i));


private static Optional<String> tryFindSimilar(String s) {
    return Optional.of("similar" + s);
}
```

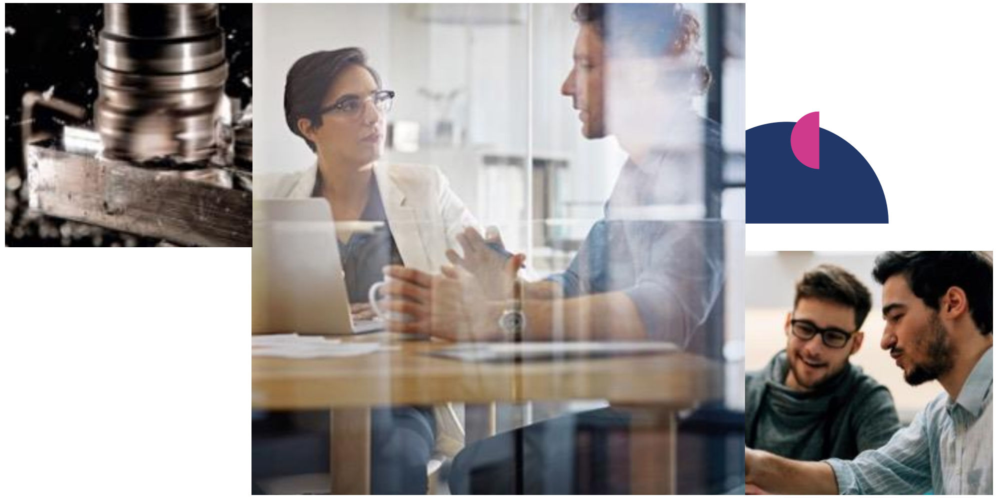
# Using Optional to prevent NullPointerException

- By changing return type of getters to Optional we take advantage of functional programming style



```
String latitude = Optional
        .ofNullable(carData)
        .flatMap(CarData::getGpsPosition)
        .flatMap(GPSPosition::getCoordinates)
        .map(Coordinates::getLatitude)
        .orElse("unknown");
```

# Thank you!



## Michał Rudnik

michal.rudnik@ttpsc.pl

Transition Technologies PSC Sp. z  o.o.

www.ttpsc.pl