Building Errorless Processing Pipelines with Optionals

José Paumard

@JosePaumard | blog.paumard.org

Agenda

Optionals

Advanced uses of Optionals

Optionals

When there is no result

The Concept of Optional

- Sometimes we do not know what the result should be

```
List<Person> people = new ArrayList<>();

int maxAge =
people.stream()
      .map(p -> p.getAge())
      .reduce(
         0, Integer::max);
```

- In this case all the ages are positive integers, so 0 is the identity element

The Concept of Optional

- But in this case:

```
List<Person> people = new ArrayList<>();

// int averageAge =
people.stream()
      .mapToInt(p -> p.getAge())
      .average();
```

- The average() method is a reduction, but it has no identity element
- So what should be the value of the average() of an empty list?

The Concept of Optional

- The answer is: an Optional

```
List<Person> people = new ArrayList<>();

OptionalInt average =
people.stream()
      .mapToInt(p -> p.getAge())
      .average();
```

- An Optional wraps a value that « may not be there »
- In other words: an Optional can be empty

How Can One Use an Optional?

- A first pattern:

```
Optional<Person> opt = ...;

if (opt.isPresent()) {
   Person p = opt.get();
} else {
   // there is nobody here...
}
```

- It is the classical way of using an optional, but we can do much better!

How Can One Use an Optional?

- A first variant of this first pattern:

```
Optional<Person> opt = ...;

Person p1 = opt.orElse(Person.getDefault());
```

- We provide a default value, returned if the optional is empty

How Can One Use an Optional?

- A first variant of this first pattern:

```
Optional<Person> opt = ...;

Person p1 = opt.orElse(Person.getDefault());

Person p2 = opt.orElseGet(() -> Person.getDefault());
```

- Instead of providing a default instance,
  we provide a way to build that default instance
- Thus saving the building of that instance, if it is not needed!

Second Type of Patterns

- There is another way of using optionals
- But before, we need to learn how to build an optional from scratch

Patterns to Build an Optional

- First, the default constructor of the `Optional` class is private
- So we cannot build an optional using `new`
- We have static methods:

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

Patterns to Build an Optional

- First, the default constructor of the `Optional` class is private
- So we cannot build an optional using new
- We have static methods:

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

Optional<String> nonEmpty = Optional.of(s); // NullPointerException
```

- Be careful: `of()` throws a `NullPointerException` if `s` is null

Patterns to Build an Optional

- First, the default constructor of the `Optional` class is private
- So we cannot build an optional using new
- We have static methods:

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

Optional<String> nonEmpty = Optional.of(s); // NullPointerException

Optional<String> couldBeEmpty = Optional.ofNullable(s);
```

- If null is passed, then the returned optional is the empty one

Second Type of Patterns

- There is another family of methods on the `Optional` class

```
public Optional<U> map(Function<T, U> mapper);
```

- Returns an empty optional if `this` is empty

Second Type of Patterns

- There is another family of methods on the `Optional` class

```
public Optional<U> map(Function<T, U> mapper);

public Optional<T> filter(Predicate<T> filter);
```

- Returns an empty optional if `this` is empty

Second Type of Patterns

- There is another family of methods on the `Optional` class

```
public Optional<U> map(Function<T, U> mapper);

public Optional<T> filter(Predicate<T> filter);

public void ifPresent(Consumer<T> consumer);
```

- Does nothing if `this` is empty

Second Type of Patterns

- This second type of patterns sees an optional as a special stream:
- That can hold only one or zero element

Advanced Use of Optional

No nulls, no exceptions, parallel computations

The NewMath Class

- Let us write a new math class!

```
public class NewMath {

   public static Optional<Double> sqrt(Double d) {
      return d > 0d ? Optional.of(Math.sqrt(d));
                      Optional.empty();
   }

   public static Optional<Double> inv(Double d) {
      return d != 0d ? Optional.of(1d/d);
                       Optional.empty();
   }
}
```

The NewMath Class

- Let us write a new math class!

```
List<Double> doubles = ...;
List<Double> result = new ArrayList<>();

doubles.stream()
       .forEach(
          d -> NewMath.sqrt(d)
                      .ifPresent(result::add);
       );
```

The NewMath Class

- Let us write a new math class!

```
List<Double> doubles = ...;
List<Double> result = new ArrayList<>();

doubles.stream()
       .forEach(
          d -> NewMath.sqrt(d)
                      .flatMap(NewMath::inv)
                      .ifPresent(result::add);
       );
```

Using `flatMap()`

- What is this `flatMap()` method?

```
Optional<U> flatMap(Function<T, Optional<U>> flatMapper);
```

- It takes the content of the optional
- If there is one it maps it
- And then returns a wrapping optional that can be empty
- It looks like the `flatMap()` method from `Stream`

Using `flatMap()`

- Would it be possible to build a function that returns a `Stream<T>`
- That would contain the value if the `Optional` is not empty
- And that would be empty if the `Optional` is itself empty?

Using `flatMap()`

- Let us take an example with the `NewMath` class

```
d -> NewMath.inv(d)                                    // Optional<Double>
```

- If d is null, then the optional is empty

Using `flatMap()`

- Let us take an example with the `NewMath` class

```
d -> NewMath.inv(d)                                  // Optional<Double>
        .flatMap(d -> NewMath.sqrt(d))               // Optional<Double>
```

- If the first optional is empty, then the second one is empty too
- If `1/d` is negative, then the second optional is empty

Using `flatMap()`

- Let us take an example with the `NewMath` class

```
d -> NewMath.inv(d)                          // Optional<Double>
        .flatMap(d -> NewMath.sqrt(d))       // Optional<Double>
        .map(d -> Stream.of(d))              // Optional<Stream<Double>>
```

- The mapping can return:
  - an empty optional
  - or an optional that holds a stream with the result of `sqrt(1/d)`

Using flatMap()

- Let us take an example with the NewMath class

```
d -> NewMath.inv(d)                          // Optional<Double>
        .flatMap(d -> NewMath.sqrt(d))       // Optional<Double>
        .map(d -> Stream.of(d))              // Optional<Stream<Double>>
```

- How can I open this Optional?

Using `flatMap()`

- Let us take an example with the `NewMath` class

```
d -> NewMath.inv(d)                              // Optional<Double>
        .flatMap(d -> NewMath.sqrt(d))           // Optional<Double>
        .map(d -> Stream.of(d))                  // Optional<Stream<Double>>
        .orElseGet(                          ) ;
```

- By using the `orElseGet()` method, we can « open » an optional
- If there is a value in it, we want to return it (it is a `Stream`!)
- If there is none, we want to return an empty `Stream`

Using flatMap()

- Let us take an example with the NewMath class

```
d -> NewMath.inv(d)                          // Optional<Double>
        .flatMap(d -> NewMath.sqrt(d))       // Optional<Double>
        .map(d -> Stream.of(d))              // Optional<Stream<Double>>
        .orElseGet(() -> Stream.empty()) ; // Stream<Double>
```

- This last call returns a Stream:
  - that holds the result
  - or is empty

Using `flatMap()`

- This is the function we were looking for!

```
Function<Double, Stream<Double>> invSqrt =
d -> NewMath.inv(d)                          // Optional<Double>
        .flatMap(d -> NewMath.sqrt(d))       // Optional<Double>
        .map(d -> Stream.of(d))              // Optional<Stream<Double>>
        .orElseGet(() -> Stream.empty()) ; // Stream<Double>
```

- If `sqrt(1/d)` cannot be computed, then this stream is empty
- If it can, then this stream just holds the value

Using `flatMap()`

- And by the way, with method references:

```
Function<Double, Stream<Double>> invSqrt =
d -> NewMath.inv(d)                    // Optional<Double>
        .flatMap(NewMath::sqrt)        // Optional<Double>
        .map(Stream::of)               // Optional<Stream<Double>>
        .orElseGet(Stream::empty) ; // Stream<Double>
```

Using `flatMap()`

- We can then write our data processing pattern

```
List<Double> doubles = ...;

List<Double> invSqrtOfDoubles =
doubles.stream()
       .flatMap(invSqrt)
       .collect(Collectors.toList()); // collects the elements in a list
```

Using `flatMap()`

- We can then write our data processing pattern

```
List<Double> doubles = ...;

List<Double> invSqrtOfDoubles =
doubles.stream().parallel()
       .flatMap(invSqrt)
       .collect(Collectors.toList()); // collects the elements in a list
```

- And this time we can safely compute this stream in parallel!

Live Coding

The NewMath example

Live Coding Summary

- We saw how to efficiently use streams and optionals together
- We saw the importance of the flatMap pattern
- We could leverage the full power of those two API:
  - on a very clean pattern
  - and very efficient too!

Summary

- Optional concept: basic patterns = an optional is a wrapper that can be empty
- But it can also be seen as a special type of Stream
- Advanced patterns built on optionals: map, filter, ifPresent, flatMap
- How to have streams and optionals play nicely together
- How to build very clean and efficient patterns