Using Optionals

Supporting Methods That Cannot Produce a Result

We already covered several uses of the `Optional` class, especially in the case where you are calling a terminal operation on a stream that does not have an identity element. This case is not easy to handle because you cannot return any value, including 0, and returning `null` would make your code have to handle `null` values in places where you would not want it.

Being able to use the `Optional` class, for the cases where you cannot produce a value, offers many opportunities for better patterns, especially for better handling of errors. This is the main reason why you should be using option objects: they signal that a method may not produce a result in certain circumstances. Storing an instance of `Optional` instance in a field, in a list, in a map, or passing it as a method argument is not what options have been created.

If you design a method that returns an optional, or need to store an optional in a variable, then you should not return null or set this variable to null. You should leverage the fact that your optional can be empty instead.

In a nutshell, the `Optional` class is a wrapper class, that can wrap a reference: `Optional<T>`, or a value: `OptionalInt`, `OptionalLong`, and `OptionalDouble`. The difference with the classic wrapper types that you already know: `Integer`, `Long`, `Double`, etc... is that an optional object can be empty. Such an instance does not wrap anything.

If you need a mechanism to return something from your method that would mean *no value* and that returning null could lead to errors, including `NullPointerException`, then you should consider returning an optional object and returning an empty optional in this case.

Creating Optional Objects

The `Optional` class is a final class with private constructor. So, the only way you have to create an instance of it is by calling one of its factory methods. There are three of them.

1. You can create an empty optional by calling `Optional.empty()`.
2. You can wrap a non-null element by calling `Optional.of()` with this element as an argument. Passing a null reference to this method is not allowed. You will get a `NullPointerException` in this case.

3. You can wrap any element by calling `Optional.ofNullable()` with this element as an argument. You may pass a null reference to this method. In this case, you will get an empty optional.

These are the only ways to create an instance of this class. As you can see, you cannot wrap a null reference with an optional object. The consequence is that, opening a nonempty optional will always return a non-null reference.

The `Optional<T>` class has three equivalent classes used with the specialized streams of numbers: `OptionalInt`, `OptionalLong`, and `OptionalDouble`. These classes are wrappers of primitive types, that is, values. The method `ofNullable()` would not make sense for these classes because a value cannot be null.

Opening an Optional Object

There are several ways of using an optional and accessing the element it wraps, if any. You can directly query the instance you have and open it if there is something in it, or you can use stream-like methods on it: `map()`, `flatMap()`, `filter()`, and even an equivalent of `forEach()`.

Opening an optional to get its content should be made with caution because it will raise a `NoSuchElementException` if the optional is empty. Unless you are sure that there is an element in your optional, you should protect this operation by first testing it.

Two methods are there for you to test your optional object: `isPresent()`, and `isEmpty()`, added in Java SE 11.

Then, to open your optional, you can use the following methods.

- `get()`: this method has been deprecated because is looks like a *getter*, but it can throw a `NoSuchElementException` if the optional is empty.
- `orElseThrow()` is the preferred pattern since Java SE 10. It does the same as the `get()` method, but its name does not leave any doubt that it can throw a `NoSuchElementException`.
- `orElseThrow(Supplier exceptionSupplier)`: does the same as the previous method. It uses the supplier you pass as an argument to create the exception that it throws.

You can also try to get the content of an optional object and provide an object that will be returned in case the optional is empty.

- `orElse(T returnedObject)`: returns the argument if called on an empty optional.
- `orElseGet(Supplier<T> supplier)`: does the same as the previous one, without having to build the returned object, in case building this object proves to be expensive. Indeed, the provided supplier is invoked only if needed.

Lastly, you can create another optional if this optional is empty.

- `or(Supplier<Optional> supplier)`: returns this unmodified optional if it is not empty and calls the provided supplier if it is. This supplier creates another optional that is returned by this method.

Processing an Optional Object

The `Optional` class also provides patterns so that you can integrate your optional objects' processing with stream processing. It has methods that directly correspond to methods from the Stream API that you can use to process your data in the same way, and that will seamlessly integrate with streams. Those methods are `map()`, `filter()`, and `flatMap()`. They take the same arguments as their twin methods from the Stream API, apart from `flatMap()`, that takes a function that returns an instance of `Optional<T>` instead of an instance of `Stream`.

These methods return an optional object with the following convention.

1. If the optional from which they are called is empty, then they return an optional object.
2. If it is not empty, then their argument, function, or predicate is applied to the content of this option. The result is wrapped in another option, which it returned by this method.

Using these methods can lead to more readable code in some stream patterns.

Suppose you have a list of `Customer` instances with an `id` property. You need to find the name of the customer with a given ID. Using the stream vocabulary, you need to *find* the customer with the given ID, and to *map* it to its name.

You can do this with the following pattern.

```
String findCustomerNameById(int id){
    List<Customer> customers = ...;

    return customers.stream()
                    .filter(customer->customer.getId() == id);
                    .findFirst()
                    .map(Customer::getName)
                    .orElse("UNKNOWN");
}
```

You can see on this pattern that the `map()` method comes from the `Optional` class, and it integrates nicely with the stream processing. You do not need to check if the optional object returned by the `findFirst()` method is empty or not; calling `map()` does in fact this for you.

Getting the Two Authors that Published the Most Together

Let us examine another, more complex example that demonstrates how to use these methods. Going through this example will show you several of the major patterns of the Stream API, the Collector API, and the optional objects.

Suppose you have a set of articles that you need to process. An article has a title, an inception year, and a list of authors. An author has a name.

There is a lot of articles in your list, and you need to know what authors have published the largest number of articles together.

Your first idea could be to build a stream of pairs of authors for a given article. This is in fact the cartesian product of the set of the authors of a given article. You do not need all the pairs in this stream. You are not

interested in the pairs where the two authors are in fact the same author; a pair of two authors (*A1*, *A2*) is the same as the pair (*A2*, *A1*). To implement this constraint, you can add a constraint to a pair, by stating that, in a pair, the authors are sorted alphabetically.

Let us write two records for this model.

```
record Article (String title, int inceptionYear, List<Author> authors) {}

record Author(String name) implements Comparable<Author> {

    public int compareTo(Author other) {
        return this.name.compareTo(other.name);
    }
}

record PairOfAuthors(Author first, Author second) {

    public static Optional<PairOfAuthors> of(Author first, Author second) {
        if (first.compareTo(second) > 0) {
            return Optional.of(new PairOfAuthors(first, second));
        } else {
            return Optional.empty();
        }
    }
}
```

Creating a factory method in the `PairOfAuthors` record allows you to control what instances of this record are allowed and prevent the creation of pairs you do not need. To show that this factory method may not be able to produce a result, you can wrap it in an optional. This perfectly respects the principle: if you cannot produce a result, return an empty optional.

Let us write a function that creates a `Stream<PairOfAuthors>` for a given article. You can make a Cartesian product with two nested streams.

As a first step, you may write a bifunction that creates this stream from an article and an author.

```
BiFunction<Article, Author, Stream<PairOfAuthors>> buildPairOfAuthors =
    (article, firstAuthor) ->
        article.authors().stream().flatMap(
            secondAuthor -> PairOfAuthors.of(firstAuthor, secondAuthor).stream());
```

This bifunction creates an optional object from the `firstAuthor` and the `secondAuthor`, taken from the stream built on the authors of the article. You can see that the `stream()` method is called on the optional object returned by the `of()` method. The returned stream is empty if the optional is empty and contains only a single pair of authors otherwise. This stream is processed by `flatMap()` method. This method opens the streams, the empty ones will just vanish, and only the valid pairs will appear in the resulting stream.

You can now build a function that uses this bifunction to create a stream of pairs of authors from an article.

```
Function<Article, Stream<PairOfAuthors>> toPairOfAuthors =
    article ->
    article.authors().stream()

```

```
                        .flatMap(firstAuthor -> buildPairOfAuthors.apply(article, firstAuthor));
```

Knowing the two authors that published to most together can be done from a histogram in which the keys are the pairs of authors and the values the number of articles they wrote together.

You can build a histogram with the `groupingBy()` collector. Let us first create the stream of pair of authors.

```
Stream<PairOfAuthors> pairsOfAuthors =
    articles.stream()
            .flatMap(toPairOfAuthors);
```

This stream is built in such a way that if a pair of authors wrote two articles together, this pair appears twice in the stream. So, what you need to do is to count how many times each pair appears in this stream. This can be done with a `groupingBy()` pattern in which the classifier is the identity function: the pair itself. At this point, the values are lists of pairs, which you need to count. So the downstream collector is just the `counting()` collector.

```
Map<PairOfAuthors, Long> numberOfAuthorsTogether =
    articles.stream()
            .flatMap(toPairOfAuthors)
            .collect(Collectors.groupingBy(
                    Function.identity(),
                    Collectors.counting()
            ));
```

Finding the authors that published the most together consists in extracting the maximum value of this map. You can create the following function for this processing.

```
Function<Map<PairOfAuthors, Long>, Map.Entry<PairOfAuthors, Long>> maxExtractor =
    map -> map.entrySet().stream()
                         .max(Map.Entry.comparingByValue())
                         .orElseThrow();
```

This function calls the `orElseThrow()` method on the optional object returned by the `Stream.max()` method.

Can this optional object be empty? For it to be empty the map itself has to be empty meaning that there were no pairs of authors in the original stream. As long as you have at least one article with at least two authors, then this optional is not empty.

Getting the Two Authors that Published the Most Together Per Year

Let us go one step further and wonder if you could do the same processing for each year. In fact, being able to implement this processing with a single collector would solve your problem because you could then pass it as a downstream collector to a `groupingBy(Article::inceptionYear)` collector.

The postprocessing of the map to extract the maximum can be made a `collectingAndThen()` collector. This pattern has already been covered in a previous section "Using a Finisher to Post-Process a Collector. This collector is the following.

Let us extract the `groupingBy()` collector and the finisher. If you are using an IDE to type this code, you can use it to get the correct types for your collector.

```
Collector<PairOfAuthors, ?, Map<PairOfAuthors, Long>> groupingBy =
        Collectors.groupingBy(
                Function.identity(),
                Collectors.counting()
        );

Function<Map<PairOfAuthors, Long>, Map.Entry<PairOfAuthors, Long>> finisher =
        map -> map.entrySet().stream()
                  .max(Map.Entry.comparingByValue())
                  .orElseThrow();
```

Now you can merge them together in a single `collectingAndThen()` collector. You can recognize the `groupingBy()` collector as the first argument and the the `finisher` function as a second argument.

```
Collector<PairOfAuthors, ?, Map.Entry<PairOfAuthors, Long>> pairOfAuthorsEntryCollector =
    Collectors.collectingAndThen(
            Collectors.groupingBy(
                Function.identity(),
                Collectors.counting()
            ),
            map -> map.entrySet().stream()
                      .max(Map.Entry.comparingByValue())
                      .orElseThrow()
    );
```

You can now write the full pattern with the initial flatmap operation and this collector.

```
Map.Entry<PairOfAuthors, Long> numberOfAuthorsTogether =
    articles.stream()
            .flatMap(toPairOfAuthors)
            .collect(pairOfAuthorsEntryCollector);
```

Thanks to the `flatMapping()` collector, you can write this code with a single collector by merging the intermediate `flatMap()` and the terminal collector. The following code is equivalent to the previous one.

```
Map.Entry<PairOfAuthors, Long> numberOfAuthorsTogether =
    articles.stream()
            .collect(
                Collectors.flatMapping(
                    toPairOfAuthors,
                    pairOfAuthorsEntryCollector));
```

Finding the two authors that published the most, per year, is just a matter of passing this `flatMapping()` collector as a downstream collector to the right `groupingBy()`.

```
Collector<Article, ?, Map.Entry<PairOfAuthors, Long>> flatMapping =
    Collectors.flatMapping(
            toPairOfAuthors,
            pairOfAuthorsEntryCollector));

Map<Integer, Map.Entry<PairOfAuthors, Long>> result =
    articles.stream()
            .collect(
```

```
                Collectors.groupingBy(
                    Article::inceptionYear,
                    flatMapping
                )
            );
```

You may remember that, deep inside this `flatMapping()` collector there is a call to the `Optional.orElseThrow()`. Checking if this call could fail was easy in the previous pattern, because having an empty optional at this point was fairly easy to guess.

Now that we have used this collector as a downstream collector, the situation is different. How can you be sure that, for each year, you have at least one article written by at least two authors? It would be safer to protect this code against any `NoSuchElementException`.

Avoiding the Opening of Optionals

What could be an acceptable pattern in the first context is much more dangerous now. Dealing with it consists of not calling `orElseThrow()` in the first place.

In that case, the collector becomes the following. Instead of creating a key-value pair of pair of authors and a long number, it wraps the result in an optional object.

```
Collector<PairOfAuthors, ?, Optional<Map.Entry<PairOfAuthors, Long>>>
        pairOfAuthorsEntryCollector =
            Collectors.collectingAndThen(
                Collectors.groupingBy(
                    Function.identity(),
                    Collectors.counting()
                ),
                map -> map.entrySet().stream()
                          .max(Map.Entry.comparingByValue())
            );
```

Note that the `orElseThrow()` is not called anymore, thus leading to an optional in the signature of the collector.

This optional also appears in the signature of the `flatMapping()` collector.

```
Collector<Article, ?, Optional<Map.Entry<PairOfAuthors, Long>>> flatMapping =
        Collectors.flatMapping(
                toPairOfAuthors,
                pairOfAuthorsEntryCollector
        );
```

Using this collector to create the map of the pair of authors per year creates a map of type `Map<Integer, Optional<Map.Entry<PairOfAuthors, Long>>>`, a type we do not need: having a map in which values are empty optionals is useless and maybe costly. It is an antipattern. Unfortunately, there is no way you can guess that this optional will be empty before computing this maximum value.

Once this intermediate map is built, you need to get rid of empty optionals to build the map that represent the histogram you need. We are going to use the same technique as previously: call the `stream()` method on the optional objects in a `flatMap()` so that the `flatMap()` operation silently removes the empty optionals.

The pattern is the following.

```
Map<Integer, Map.Entry<PairOfAuthors, Long>> histogram =
    articles.stream()
            .collect(
                Collectors.groupingBy(
                        Article::inceptionYear,
                        flatMapping
                )
            )  // Map<Integer, Optional<Map.Entry<PairOfAuthors, Long>>>
            .entrySet().stream()
            .flatMap(
                entry -> entry.getValue()
                              .map(value -> Map.entry(entry.getKey(), value))
                              .stream())
            .collect(Collectors.toMap(
                    Map.Entry::getKey, Map.Entry::getValue
            )); // Map<Integer, Map.Entry<PairOfAuthors, Long>>
```

Note the flatmap function in this pattern. It takes an `entry`, whose value is of type `Optional<Map.Entry<PairOfAuthors, Long>>` as an argument, and calls `map()` on this optional.

If the optional is empty, this call returns an empty optional. The mapping function is then ignored. The next call to `stream()` then returns an empty stream, which will be removed from the main stream because we are in a `flatMap()` call.

If there is a value in the optional, then the mapping function is called with this value. This mapping function creates a new key-value pair with the same key and this existing value. This key-value pair is of type `Map.Entry<PairOfAuthors, Long>`, and it is wrapped in an optional object by this `map()` method. The call to `stream()` makes a stream with the content of this optional, which is then opened by the `flatMap()` call.

This patterns maps a `Stream<Map.Entry<Integer, Optional<Map.Entry<PairOfAuthors, Long>>>>` with empty optionals to a `Stream<Map.Entry<Integer, Map.Entry<PairOfAuthors, Long>>>`, removing all the key/value pairs that had empty optionals.

Recreating the map can safely be done with a `toMap()` collector, because you know that you cannot have the same key twice in this stream.

This pattern uses three important points of the Stream API and the optionals.

1. The `Optional.map()` method that returns an empty optional if it is called on an empty optional.
2. The `Optional.stream()` method that opens a stream on the content of an optional. If the optional is empty, then the returned stream is also empty. It allows you to move from the optional space to the stream space seamlessly.
3. The `Stream.flatMap()` method that opens the streams built from the optionals, silently removing the empty streams.

Consuming the Content of an Optional

The `Optional` class has also two methods that take a consumer as an argument.

- `ifPresent(Consumer consumer)`: this method calls the provided consumer with the content of this optional, if any. It is in fact equivalent to the `Stream.forEach(Consumer)` method.
- `ifPresentOrElse(Consumer consumer, Runnable runnable)`: this method does the same as the previous one if the optional is not empty. If it is, then it calls the provided instance of `Runnable`.

Stating Some Rules to Use Optionals Properly

Rule #1 Never use null for an optional variable or returned value.

Rule #2 Never call `orElseThrow()` or `get()` unless you are sure the optional is not empty.

Rule #3 Prefer alternatives to `ifPresent()`, `orElseThrow()`, or `get()`.

Rule #4 Do not create an optional to avoid testing for the nullity of a reference.

Rule #5 Do not use optional in fields, method parameters, collections, and maps.

Rule #6 Do not use identity-sensitive operations on an optional object, such as reference equality, identity hash code, and synchronization.

Rule #7 Do not forget that optional objects are not serializable.

Last update: *September 14, 2021*

Implementing the Collector Interface ← **Using Optionals** → Parallelizing Streams

Home > Tutorials > The Stream API > Using Optionals