

Lesson-9- Class Notes

Java Stream API Reference:

<https://docs.oracle.com/en%2Fjava%2Fjavase%2F21%2Fdocs%2Fapi%2F%2F/java.base/java/util/stream/package-summary.html#StreamOps>

→ Fork Join framework used for Parallel Processing of Streams

→ Two types of operations on Streams

- Intermediate → Lazy
- Terminal → Once you invoke the terminal, then only your intermediate operations start functioning.

// Imperative Style

```
Button ob = new Button("Save");
```

// Lambda

```
Function<String, Button> ob = (str) -> new Button(str);
```

// Method Reference

```
Function<String, Button> ob = Button::new;
```

How Arrays can work with Stream library

Convert your input into Stream object using `Arrays.stream(input array)` and retrieve as Stream.

// Converting Array to Stream

```
int[] arr = {1,2,3,4,5,6};  
  
IntStream nums = IntStream.of(arr);  
  
int total = nums.sum();  
  
  
Item[] col = new Item[5];  
  
Stream<Item> items = Arrays.stream(col);
```

Slide-27

```
// String[] coll = strings.stream().toArray(); // Error  
// Throw Exception, not passing arguments  
// String[] coll = (String[]) strings.stream().toArray();  
// Recommended way - Convert List<String> to String[]  
String[] vals = strings.stream().toArray(String[]::new);  
System.out.println(Arrays.toString(vals));
```

```
// Using a lambda expression to create the array
String[] coll = strings.toArray(size -> new String[size]);
System.out.println(Arrays.toString(coll));
```

Summary points

- You can create streams from collections, arrays, generators, or iterators.
- Use filter to select elements and map to transform elements.
- Other operations for transforming streams include limit, distinct, and sorted.
- To obtain a result from a stream, use a reduction operator such as count, max, min, findFirst, or findAny. Some of these methods return an Optional value.
- The Optional type is intended as a safe alternative to working with null values. To use it safely, take advantage of the ifPresent and orElse methods.
- You can collect stream results in collections, arrays, strings, or maps.
- There are specialized streams for the primitive types int, long, and double such as IntStream, LongStream, and DoubleStream.
- You need to be familiar with a small number of functional interfaces in order to use the stream library.

Q-Reilly - Text Book about State and Stateless Intermediate operations.

There are stateless operations, such as filter(), map(), and flatMap(), which do not keep data around (do not maintain state) while moving from processing from one stream element to the next. They do not depend on the order or values of the elements in the stream.

And there are stateful intermediate operations, such as distinct(), limit(), skip() and sorted(), which may pass the state from previously processed elements to the processing of the next element. They require some form of state tracking or buffering to ensure their correct operation.

Stateful Example 1: The distinct() operation removes duplicate elements from the stream, ensuring that only unique elements remain. To achieve this, it needs to maintain a set, to check whether a new element is a duplicate or not.

Stateful Example 2: The limit() operation restricts the number of elements in a stream to a specified maximum size. It relies on maintaining an internal counter to keep track of how many elements have been processed. The order of elements matters because it determines which elements are included in the output.

Stateful Example 3: The sorted() operation orders the elements in the stream based on their natural order or using a custom comparator. To perform sorting, it needs to collect and buffer all the elements before sorting them. The order of elements is significant because it determines the sorting outcome.

In a stateless operation, each element is processed independently of the others. In a stateful operation, the processing of an element may depend on aspects of the other elements.

The difference between stateless and stateful operations is especially important when parallel processing of a stream is desired because a stateful operation may require more than one pass to complete.

Stateless operations, such as filter and map, retain no state from previously seen element when processing a new element -- each element can be processed independently of operations on other elements. Stateful operations, such as distinct and sorted, may incorporate state from previously seen elements when processing new elements.

Java 9 Features

`Optional::stream`

This method can be used to transform a `Stream` of optional elements to a `Stream` of present value elements:

If a value is present, returns a sequential `Stream` containing only that value, otherwise returns an empty `Stream`.

It offers a straightforward way to filter out null values.

```
List<Optional<String>> optionalList = Arrays.asList(
    Optional.of("Hello"),
    Optional.empty(),
    Optional.of("World"),
    Optional.empty()
);

List<String> filteredList = optionalList.stream()
    .flatMap(Optional::stream)
    // .flatMap(opt->opt.stream())
    .collect(Collectors.toList());

System.out.println(filteredList);
```

Output:

[Hello, World]

Refer: MainPerson.java from your Daily Class notes of Lesson 9.

Operation	Type	Description
Distinct	Intermediate	Returns a stream consisting of the distinct elements of this stream. Elements <code>e1</code> and <code>e2</code> are considered equal if <code>e1.equals(e2)</code> returns true.
filter	Intermediate	Returns a stream consisting of the elements of this stream that match the specified predicate.
flatMap	Intermediate	Returns a stream consisting of the results of applying the specified function to the elements of this stream. The function produces a stream for each input element and the output streams are flattened. Performs one-to-many mapping.
limit	Intermediate	Returns a stream consisting of the elements of this stream, truncated to be no longer than the specified size.
map	Intermediate	Returns a stream consisting of the results of applying the specified function to the elements of this stream. Performs one-to-one mapping.
peek	Intermediate	Returns a stream whose elements consist of this stream. It applies the specified action as it consumes elements of this stream. It is mainly used for debugging purposes.
skip	Intermediate	Discards the first <code>n</code> elements of the stream and returns the remaining stream. If this stream contains fewer than <code>n</code> elements, an empty stream is returned.
sorted	Intermediate	Returns a stream consisting of the elements of this stream, sorted according to natural order or the specified <code>Comparator</code> . For an ordered stream, the sort is stable.
allMatch	Terminal	Returns true if all elements in the stream match the specified predicate, false otherwise. Returns true if the stream is empty.
anyMatch	Terminal	Returns true if any element in the stream matches the specified predicate, false otherwise. Returns false if the stream is empty.
findAny	Terminal	Returns any element from the stream. An empty <code>Optional</code> object is for an empty stream.
findFirst	Terminal	Returns the first element of the stream. For an ordered stream, it returns the first element in the encounter order; for an unordered stream, it returns any element.
noneMatch	Terminal	Returns true if no elements in the stream match the specified predicate, false otherwise. Returns true if the stream is empty.
forEach	Terminal	Applies an action for each element in the stream.
reduce	Terminal	Applies a reduction operation to computes a single value from the stream.

Optional map and flatMap

Reading: Java 8 in Action Text book: Chapter -10

Imagine you have the following nested object structure for a person owning a car and having car insurance.

Listing 10.1. : The Person/Car/Insurance data model

```
public class Person {  
    private Car car;  
    public Car getCar() {  
        return car;  
    }  
}  
  
public class Car {  
    private Insurance insurance;  
    public Insurance getInsurance() {  
        return insurance;  
    }  
}  
  
public class Insurance {  
    private String name;  
    public String getName() {  
        return name;  
    }  
}
```

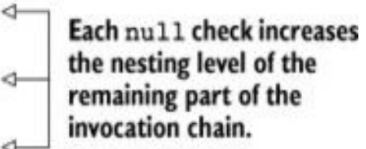
Then, what's possibly problematic with the following code?

```
public String getCarInsuranceName(Person person) {  
    return person.getCar().getInsurance().getName();  
}
```

This code looks pretty reasonable, but many people don't own a car. So what's the result of calling the method `getCar`? A common unfortunate practice is to return the null reference to indicate the absence of a value, here to indicate the absence of a car. As a consequence, the call to `getInsurance` will return the insurance of a null reference, which will result in a `NullPointerException` at run-time and stop your program from running further. But that's not all. What if person was null? What if the method `getInsurance` returned null too?

Listing 10.2. Null-safe attempt 1: deep doubts

```
public String getCarInsuranceName(Person person) {  
    if (person != null) {  
        Car car = person.getCar();  
        if (car != null) {  
            Insurance insurance = car.getInsurance();  
            if (insurance != null) {  
                return insurance.getName();  
            }  
        }  
    }  
    return "Unknown";  
}
```

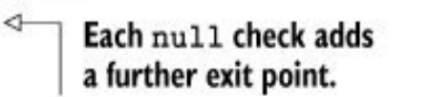


Each null check increases the nesting level of the remaining part of the invocation chain.

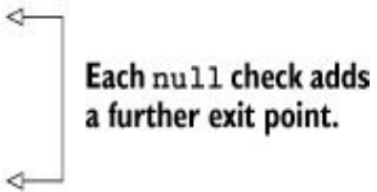
This method performs a null check every time it dereferences a variable, returning the string “Unknown” if any of the variables traversed in this dereferencing chain is a null value. The only exception to this is that you’re not checking to see if the name of the insurance company is null because, like any other company, you *know* it must have a name. Note that you can avoid this last check only because of your knowledge of the business domain, but that isn’t reflected in the Java classes modeling your data.

Listing 10.3. Null-safe attempt 2: too many exits

```
public String getCarInsuranceName(Person person) {  
    if (person == null) {  
        return "Unknown";  
    }  
    Car car = person.getCar();  
    if (car == null) {  
        return "Unknown";  
    }  
    Insurance insurance = car.getInsurance();  
    if (insurance == null) {  
        return "Unknown";  
    }  
    return insurance.getName();  
}
```



Each null check adds a further exit point.



Each null check adds a further exit point.

In this second attempt, you try to avoid the deeply nested if blocks, adopting a different strategy: every time you meet a null variable, you return the string “Unknown.” Nevertheless, this solution is also far from ideal; now the method has four distinct exit points, making it hardly maintainable. Even worse, the default value to be returned in case of a null, the string “Unknown,” is repeated in three places—and hopefully not misspelled! Of course, you may wish to extract it into a constant to avoid this problem. Furthermore, it’s an error-prone process; what if you forget to check that one property could be null? We argue in this chapter that using null to represent the absence of a value is the wrong approach. What you need is a better way to model the absence and presence of a value.

Listing 10.4. Redefining the Person/Car/Insurance data model using Optional

```
public class Person {  
    private Optional<Car> car;  
    public Optional<Car> getCar() { return car; }  
}  
  
public class Car {  
    private Optional<Insurance> insurance;  
    public Optional<Insurance> getInsurance() { return insurance; }  
}  
  
public class Insurance {  
    private String name;  
    public String getName() { return name; }  
}
```

← A person might or might not own a car, so you declare this field Optional.

← A car might or might not be insured, so you declare this field Optional.

← An insurance company must have a name.

Note how the use of the Optional class enriches the semantics of your model. The fact that a person references an Optional<Car>, and a car an Optional<Insurance>, makes it explicit in the domain that a person *might or might not* own a car, and that car *might or might not* be insured.

10.3.1. Creating Optional objects

The first step before working with Optional is to learn how to create optional objects! There are several ways.

Empty optional

As mentioned earlier, you can get hold of an empty optional object using the static factory method Optional.empty():

```
Optional<Car> optCar = Optional.empty();
```

Optional from a non-null value

You can also create an optional from a non-null value with the static factory method `Optional.of`:

```
Optional<Car> optCar = Optional.of(car);
```

If `car` were null, a `NullPointerException` would be immediately thrown (rather than getting a latent error once you try to access properties of the `car`).

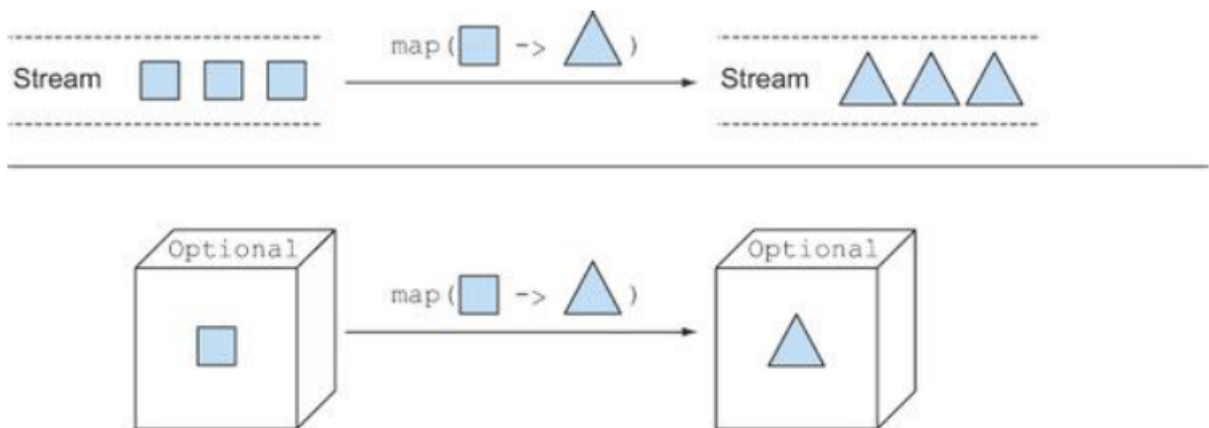
Optional from null

Finally, by using the static factory method `Optional.ofNullable`, you can create an `Optional` object that may hold a null value:

```
Optional<Car> optCar = Optional.ofNullable(car);
```

If `car` were null, the resulting `Optional` object would be empty.

Figure 10.2. Comparing the `map` methods of Streams and Optionals



This looks useful, but how could you use this to write the previous code, which was chaining several method calls in a safe way?

```
public String getCarInsuranceName(Person person) {  
    return person.getCar().getInsurance().getName();  
}
```

We have to look at another method supported by `Optional` called `flatMap`!

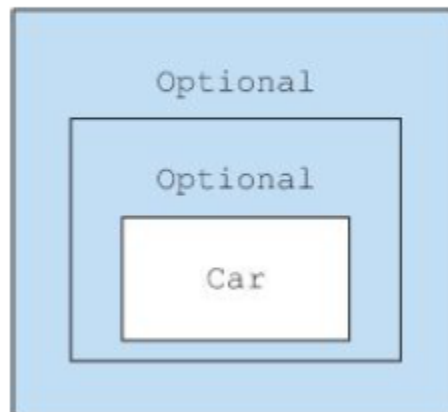
10.3.3. Chaining Optional objects with flatMap

Because you've just learned how to use map, your first reaction may be that you can rewrite the previous code using map as follows:

```
Optional<Person> optPerson = Optional.of(person);
Optional<String> name = optPerson.map(Person::getCar)
    .map(Car::getInsurance)
    .map(Insurance::getName);
```

Unfortunately, this code doesn't compile. Why? The variable `optPerson` is of type `Optional<Person>`, so it's perfectly fine to call the `map` method. But `getCar` returns an object of type `Optional<Car>` (as presented in [listing 10.4](#)). This means the result of the `map` operation is an object of type `Optional<Optional<Car>>`. As a result, the call to `getInsurance` is invalid because the outermost optional contains as its value another optional, which of course doesn't support the `getInsurance` method. [Figure 10.3](#) illustrates the nested optional structure you'd get.

Figure 10.3. A two-level optional

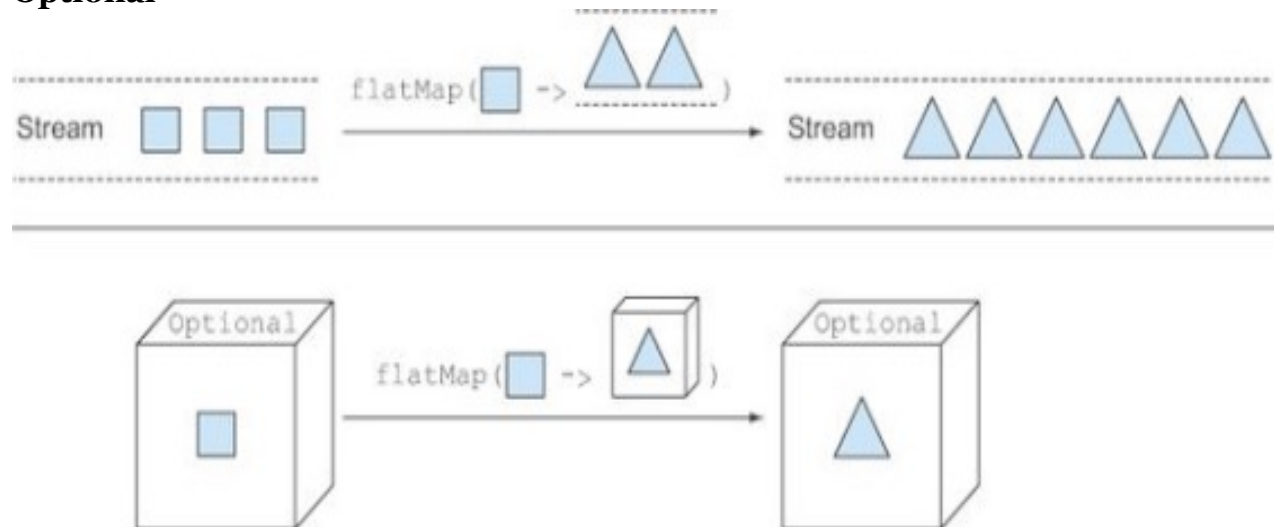


So how can we solve this problem? Again, we can look at a pattern you've used previously with streams: the `flatMap` method. With streams, the `flatMap` method takes a function as an argument, which returns another stream. This function is applied to each element of a stream, which would result in a stream of streams. But `flatMap` has the effect of replacing each generated stream by the contents of that stream. In other words, all the separate streams that are generated by the function get amalgamated or

flattened into a single stream. What you want here is something similar, but you want to flatten a two-level optional into one.

Like [figure 10.2](#) for the map method, [figure 10.4](#) illustrates the similarities between the flatMap methods of the Stream and Optional classes.

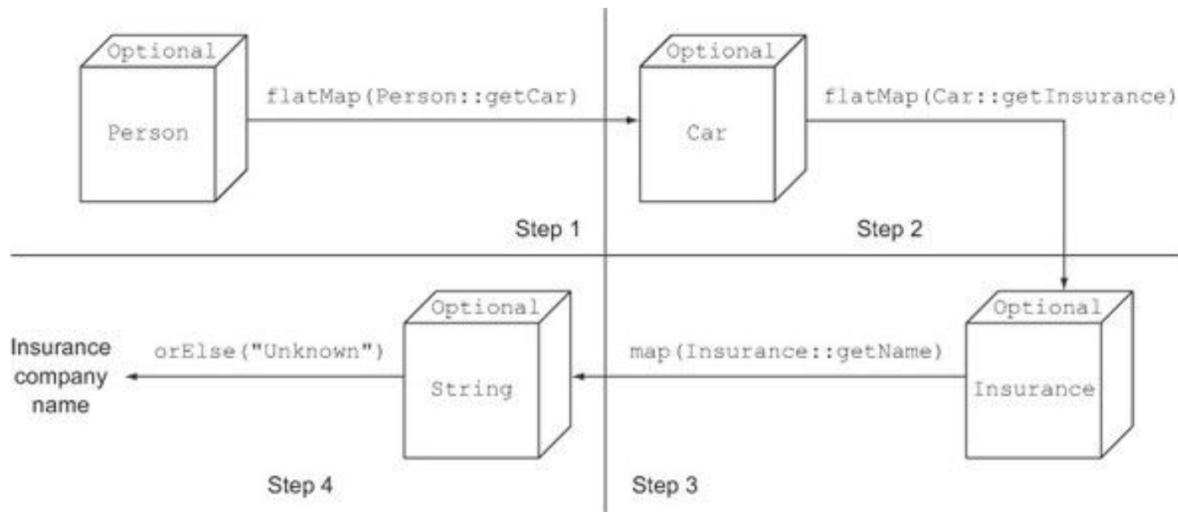
Figure 10.4. Comparing the flatMap methods of Stream and Optional



Listing 10.5. Finding a car's insurance company name with Optionals

```
public String getCarInsuranceName(Optional<Person> person) {  
    return person.flatMap(Person::getCar)  
        .flatMap(Car::getInsurance)  
        .map(Insurance::getName)  
        .orElse("Unknown");  
}
```

A default value if the resulting
Optional is empty



Here you begin with the optional wrapping the `Person` and invoking `flatMap(Person::getCar)` on it. As we said, you can logically think of this invocation as something that happens in two steps.

In step 1, a Function is applied to the `Person` inside the optional to transform it. In this case, the Function is expressed with a method reference invoking the method `getCar` on that `Person`. Because that method returns an `Optional<Car>`, the `Person` inside the optional is transformed into an instance of that type, resulting in a two-level optional that's flattened as part of the `flatMap` operation. From a theoretical point of view, you can think of this flattening operation as the operation that combines two optionals, resulting in an empty optional, if at least one of them is empty. What happens in reality is that if you invoke `flatMap` on an empty optional, nothing is changed, and it's returned as is. Conversely, if the optional wraps a `Person`, the Function passed to the `flatMap` method is applied to that `Person`. Because the value produced by that Function application is already an optional, the `flatMap` method can return it as is.

The second step is similar to the first one, transforming the `Optional<Car>` into an `Optional<Insurance>`. Step 3 turns the `Optional<Insurance>` into an `Optional<String>`: because the `Insurance.getName()` method returns a `String`, in this case a `flatMap` isn't necessary. At this point the resulting optional will be empty if any of the methods in this invocation chain returns an empty optional or will contain the desired insurance company name otherwise. So how do you read that value? After all, you'll end up getting an `Optional<String>` that may or may not contain the name of the insurance company. In [listing 10.5](#), we used another method called `orElse`, which provides a default value in case the optional is empty. There are many methods to provide default actions or unwrap an optional.