Java 8 training

`Optional` class and Streams API

Overview

- `Optional` class
- Streams API
- Collectors

Optional

- Utility class for improving the processing of potential `null` refs → avoid NPEs
- Used to:
  - Apply a processing if the wrapped value is present (is not `null`)
  - Do a different processing if it's not present (it is `null`)
    - Throw an exception, if needed

```
Optional<String> optionalValue = Optional.ofNullable(nullableString);
int result = optionalValue.map(value -> Integer.parseInt(value))
                          .orElse(0);

int nonZero = optionalValue.map(value -> Integer.parseInt(value))
                           .orElseThrow(() -> …);
```

`Optional` objects - recommended usage

Recommended usage:

1. Method return values
   a. Avoid returning `null` values → code smell which leads to ambiguity
   b. Allows the caller to chain further processing on the result

2. Inside methods, wrapping potentially `null` values
   a. Can be created using the static methods:
      i. `.ofNullable(value)` → wraps a potentially nullable value
      ii. `.of(value)` → wraps a non-`null` value
      iii. `.empty()` → returns an empty `Optional`

`Optional` objects - pay attention

The `Optional` class is:

- non `Serializable`
- `final`

→ careful when using `Optional` as:

- Method parameters
- Class fields

`Optional` usage modes

1. `.map()` + `.orElse()` / `.orElseGet()` - apply a processing if the value exists, return another value otherwise

```
int unWrapped = optional.map(Integer::parseInt)
                        .orElse(0);
                // .orElseGet(() -> new Random().nextInt());
```

2. `.map()` + `.orElseThrow()` - apply a processing if the value exists, throw an **unchecked** exception otherwise

```
int unWrappedOrThrow = optional.map(Integer::parseInt)
                               .orElseThrow(
                        () -> new RuntimeException("Nope"));
```

`.orElse()` vs `.orElseGet()`

- `Optional` has two '`orElse`' methods:
  - `orElse()`: returns the value if it's present, otherwise returns the `else` value
  - `orElseGet()`: returns the value if it's present, otherwise invokes a `Supplier` and returns the result of its invocation
- Differences:
  - '`orElse()`' - evaluated *even if the `Optional` wrapped value is present*
    - May make a difference, especially if '`orElse()`' invokes a resources expensive method
  - '`orElseGet()`' has a `Supplier` as param → evaluated *only if the wrapped value is not* present

→ Using '`orElseGet()`' may save some additional processing time

`Optional` usage modes - continued

3. `.ifPresent()` - apply a Consumer to the unwrapped value, if it exists

```
optional.ifPresent(it -> System.out.println("Value: " + it));
```

4. `.isPresent() + .get()` - verify if the value is present, get the wrapped value using `.get()` if it exists

```
if (optional.isPresent()) {
    final String unWrapped = optional.get();
} else {
    // other operations
}
```

!!! Calling `.isPresent()` before `.get()` is ***mandatory***

`Optional` usage modes - continued

5. `.flatMap()` - return an `Optional` value if the wrapped value exists

```
Optional<Section> tablets = optionalStore.flatMap(store ->
                                                  getTablets(store));

Optional<Set<Product>> optionalProducts = getProducts(tablets);
```

Can be used as chained set of `flatMap` calls, for a cleaner / monadic processing:

```
optionalProducts.flatMap(products -> getAppleTablet(products))
                .flatMap(product -> product.getDiscount())
                                                  // it may not have one
```

Hands-on - s02e01 - `Optional` in practice

Optional refactoring example

```
if (this && that && … & that) {
    // do this
}
```

```
public Date getPremiumPaymentDate() {
    FxDigitalOption fxDigitalOption = getFxDigitalOption();
    return ofNullable(fxDigitalOption).flatMap(fxDigitalOption1 -> ofNullable(fxDigitalOption1.getPremium()))
                                      .flatMap(premiums -> ofNullable(premiums[0]))
                                      .flatMap(premium -> ofNullable(premium.getPaymentDate()))
                                      .flatMap(paymentDate -> ofNullable(paymentDate.getAdjustableDate()))
                                      .flatMap(adjustableDate -> ofNullable(adjustableDate.getUnadjustedDate()))
                                      .flatMap(unadjustedDate -> ofNullable(unadjustedDate.getValue()))
                                      .map(ISO8601Date::toDate)
                                      .orElse(null);
    /*
    return (fxDigitalOption != null)
            && (fxDigitalOption.getPremium() != null)
            && (fxDigitalOption.getPremium().length > 0)
            && (fxDigitalOption.getPremium()[0] != null)
            && (fxDigitalOption.getPremium()[0].getPaymentDate() != null)
            && (fxDigitalOption.getPremium()[0].getPaymentDate().getAdjustableDate() != null)
            && (fxDigitalOption.getPremium()[0].getPaymentDate().getAdjustableDate().getUnadjustedDate() != null)
            fxDigitalOption.getPremium()[0].getPaymentDate().getAdjustableDate().getUnadjustedDate().getValue() != null) ?
            : null;
    */
}
```

Streams API

- **Stream** - sequence of elements, on which one+ operations can be applied
  - **Monad** - structure that represent computations defined as sequences of steps
  - Not related to the (old) I/O streams

- Examples:

```
stringsList.stream().forEach(System.out::println);

stringsList.stream()
           .filter(value -> isValid(value))   // → Predicate
           .count();    // counting the values which fulfill the condition
```

Stream API characteristics

- **Stream pipeline:**
  - A sequence of operations applied on a series of elements
  - Not a data structure
- Immutable data processing → new objects are returned
- Consumable → only *one* sequence of operations is allowed on a stream
  - The stream ***must*** be re-obtained if another processing is needed
- Potentially unbounded / infinite → no need for a finite size
  - Well suited for reactive streams & database queries (since JPA 2.2)

Stream pipeline

A sequence of operations which have:

- A *source*:
  - A Collection / array
  - A generator function
  - An infinite sequence generator
  - An I/O channel
  - A database query result (JPA 2.2 / Hibernate 5.2+)
- *0+ intermediate operations*
- 1 *terminal operation*

Stream operation types

Two types of operations:

- Intermediate:
  - Lazy processed → computed when the result is necessary
  - Return another Stream → *functional* nature
- Terminal:
  - Eagerly processed → immediately
  - Return the specified type

! The processing begins *only* when the terminal operation is executed

Hands-on

Trying the `Stream` operations

Intermediate operations

- `filter()` - filters the stream elements based on a condition `(Predicate)`
- `map()` - converts the input into (another / same) type `(Function)`
- `flatMap()` - converts the input into a stream of (other) values `(Function)`
- `distinct()` - returns the distinct values of the stream
- `sorted()` - natural / specific order sorting `(Comparator)`
- `peek()` - performs the provided operation on each item `(Consumer)`
- `limit()` - limits the returned values
- `skip()` - skips the first n values

} pagination support

Terminal operations

- `forEach()` - applies an operation on each item `(Consumer)`
- `forEachOrdered()` - applies an operation on each ordered item `(Consumer)`
- `toArray()` - returns the array of items
- `reduce()` - performs a reduction operation on the elements `(BinaryOperator)`
- `collect()` - collects the results into a new stream source `(Collector)`
- `min()` - returns the minimum stream item `(Comparator)`
- `max()` - returns the maximum stream item `(Comparator)`
- `count()` - counts the stream items

- `anyMatch()` - specifies if there's any matching item `-> Predicate`
- `allMatch()` - specifies if all the items are matching `-> Predicate`
- `noneMatch()` - specifies if no items are matching `-> Predicate`

- `findFirst()` - returns the first matching value from the stream `-> Optional`
- `findAny()` - returns any matching value from the stream `-> Optional`

Intermediate operations and their state

- By default, most intermediate operations are stateless → they do not maintain any state, to allow:
  - Parallelizing the stream processing (further presented)
  - Optimizing the stream processing → loop fusion (further presented)
- Some operations are inherently stateful:
  - `distinct()`
  - `sorted()`
  - `limit()`
  - `skip()`

They need to use the state of previously processed elements when processing the current element

Stateless → stateful operations

- Some stateless stream operations can become stateful if they alter a common object:

```
Set<Integer> lengths = new HashSet<>();
strings.stream()
       .map(it -> { if (lengths.add(it.length())) return 0;
                    else return 1; });
```

- The problem with state → may render non-deterministic behavior
- The solution: avoid using stateful operations, as much as possible → refactor your code to use stateless operations

Loop fusion & short circuiting

- **Loop fusion** - streams processing optimization, which leads to a vertical verification of the intermediate stream operations
- Optimizes the (default) sequential processing of the elements, so that the operations will be processed only if they are needed

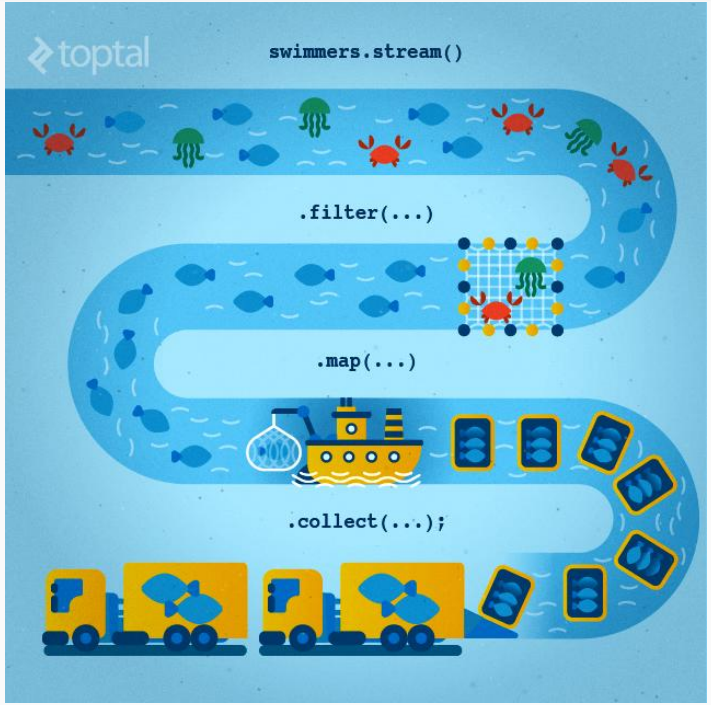
```
List<String> strings = Arrays.asList("Testing", "loop", "fusion");
int firstLongerThan4 = strings.stream()
                              .filter(s -> s.length() > 4)
                              .mapToInt(String::length)
                              .findFirst()
                              .orElse(0);
```

`findFirst()` → the stream processing doesn't need to process all the elements → **short-circuiting** nature

Map / reduce

- **Map / reduce** - splitting the input data into independent (mapping) tasks, followed by a reduction operation
- Can be:
  - Sequential
  - Parallel
- Mapping - converting:
  - From a type to another type - `.map()`
  - From a stream to a ‘flattened’ type - `.flatMap()`
- Reduction operations:
  - `.collect()`, `.average()`, `.count()`, `.reduce()`, `.sum()`

Visual map / reduce (© toptal.com)



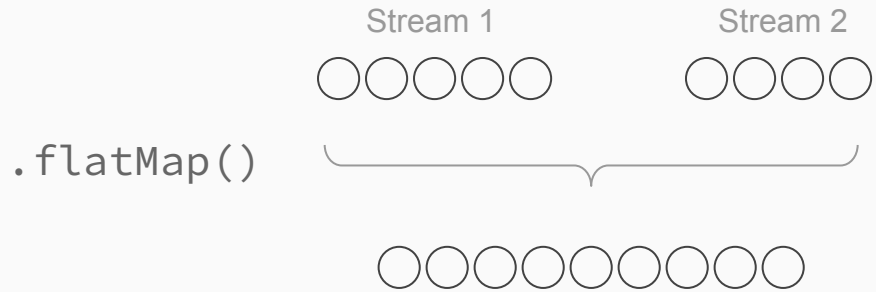
Reduction operations

- **Reduction** - returning one value by combining the content of a stream
- Reduction operations:
  - `average`
  - `collect`
  - `count`
  - `min`
  - `max`
  - `sum`
  - `reduce`

http://docs.oracle.com/javase/tutorial/collections/streams/reduction.html

Flat mapping

- **Flat mapping** - mapping an array / collection of streams into a single stream
- Can be seen as a **union** of streams



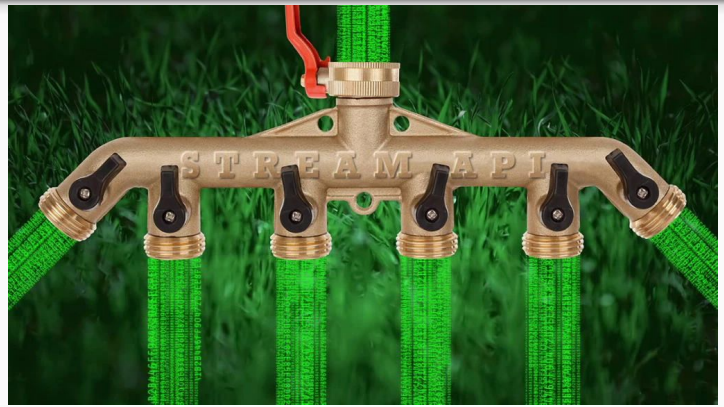
`flatMap` (for streams and `Optional`)

- For **streams**: returns a stream from an array / collection of streams

```
Stream<String[]> strings      → .flatMap → Stream<String>
Stream<List<String>> items    → .flatMap → Stream<String>
Stream<Set<String>> distinct  → .flatMap → Stream<String>
```

- For **`Optional`**: returns an `Optional<U>` from an `Optional<T>`

Streams parallelism



- Streams can be processed:
  - Sequentially - using `.stream()`
  - In parallel - using `.parallelStream()`
- Parallel streams:
  - Can* greatly improve the processing speed
  - Backed by the `ForkJoin common` thread pool
    - Inherit the forking / joining benefits of the `ForkJoin` pool
    - Changing the parallelism level (threads count):

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=4
```

When to use parallel streams

Consider S.**parallelStream()**.operation(O) instead of S.**stream()**.operation(O) if:

- The applied operation (O) is:
  - *Independent* → the element's computation is *stateless* (does not rely / impact on other elem)
  - Either / or / both:
    - *Computationally expensive*
    - Applied to *many elements of efficiently splittable data structures*
- The source collection (S) is *efficiently splittable*:
  - The most efficiently splittable sources: `ArrayLists`, `{Concurrent}HashMaps`, arrays
  - The least efficient: `LinkedLists`, `BlockingQueues` and most IO-based sources

Parallel streams & elements number

The parallelization speed impr. greatly depend on the streamed elements number

- For some ops (`min()`, `max()`) - speed improvements are seen around 10k elements
  - Bigger datasets can benefit from >20x improvement
- The worst slowdowns - when there are less than 100 elements

Source & more details:

- Stream Parallel Guidance
- Parallel streams in Java: Benchmarking and performance considerations

Creating parallel streams

Two main ways:

1. `Stream.parallelStream()`
2. `StreamSupport` class - specify if the stream is parallel / sequential
   a. Generic streams
   b. Int, long & double streams

Extras:

- ForkJoin vs parallel streams vs `ExecutorService`
- Reducing Streams load with `AirConcurrentMap`

Streams API performance

- Motivation for inventing streams – *making parallelism more accessible to developers*
  - Source: Brian Goetz - State of the lambda
- Streams performance vs loop performance:
  - On average: streams performance <= loop performance
  - Depends on several aspects - warm-up, parallelism, additional processing
- Main reasons to use the Streams API:
  - Functional composition of the code
  - Easier parallelization
  - Promoting stateless operations and immutable results

`Collectors` class

Static methods for many terminal operations:

- `toSet, toMap, toList, toCollection`
- `averaging`
- `grouping`
- `joining`
- `partitioning`
- `reducing`
- `summing`
- `summarizing`

IntStream, DoubleStream, LongStream

- Stream extending interfaces for streams of <T extends Number> objects
- Apply stream / map / reduce / count operations on the resulted streams

`java.util.Map` hierarchy

- `Map` doesn't support streams (directly)
  - Streams are supported on keys, values & entries
- Many useful methods have been added in Java 8
  - `.compute, .computeIfPresent, .computeIfAbsent`
  - `.getOrDefault`
  - `.merge`
  - `.putIfAbsent`
- All the methods use lambda expressions as parameters

Hands-on - using the `Map` streams

Streams references

Streams summary - https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html

Java Stream debugger - https://plugins.jetbrains.com/plugin/9696-java-stream-debugger

Q&A session

1. You ask, I answer
2. I ask, you answer