# Optional in Java 8 cheat sheet

## Optional in Java 8 cheat sheet

`java.util.Optional<T>` in Java 8 is a poor cousin of `scala.Option[T]` and `Data.Maybe` in Haskell. But this doesn't mean it's not useful. If this concept is new to you, imagine `Optional` as a container that may or may not contain some value. Just like all references in Java can point to some object or be `null`, `Option` may enclose some (non-null!) reference or be empty.

Turns out that the analogy between `Optional`and nullable references is quite sensible.`Optional` was introduced in Java 8 so obviously it is not used throughout the standard Java library - and never will be for the backward compatibility reasons. But I recommend you at least giving it a try and using it whenever you have nullable references. `Optional` instead of plain `null` is statically checked at compile time and much more informative as it clearly indicates that a given variable may be present or not. Of course it requires some discipline - you should never assign `null` to any variable any more.

Usage of *option* (*maybe*) pattern is quite controversial and I am not going to step into this discussion. Instead I present you with few use-cases of `null` and how they can be retrofitted to `Optional<T>`. In the following examples given variables and types are used:

```
1  publicvoidprint(String s) {
2      System.out.println(s);
3  }
4
5  String x = //...
6  Optional<String> opt = //...
```

`x` is a String that *may* be `null`, `opt` is never `null`, but may or may not contain some value (*present* or *empty*). There are few ways of creating `Optional`:

```
1  opt = Optional.of(notNull);
2
3  opt = Optional.ofNullable(mayBeNull);
4
5  opt = Optional.empty();
```

In the first case `Optional` *must* contain not `null` value and will throw an exception if `null` is passed. `ofNullable()` will either return empty or present (set) `Optional.empty()` always return empty `Optional`, corresponding to `null`. It's a singleton because `Optional<T>` is immutable.

## `ifPresent()` - do something when `Optional` is set

Tedious `if` statement:

```
1  if(x != null) {
2      print(x);
3  }
```

can be replaced with higher-order function `ifPresent()`:

```
1  opt.ifPresent(x -> print(x));
2  opt.ifPresent(this::print);
```

The latter syntax (method reference) can be used when lambda argument (`String x`) matches function formal parameters.

## `filter()` - reject (filter out) certain `Optional` values.

Sometimes you want to perform some action not only when a reference is set but also when it meets certain condition:

```
1  if(x != null&& x.contains("ab")) {
2      print(x);
3  }
```

This can be replaced with `Optional.filter()` that turns present (set) `Optional` to empty `Optional` if underlying value does not meet given predicate. If input`Optional` was empty, it is returned as-is:

```
1  opt.
2     filter(x -> x.contains("ab")).
3     ifPresent(this::print);
```

This is equivalent to more imperative:

```
if(opt.isPresent() && opt.get().contains("ab")) {
    print(opt.get());
}
```

## `map()` - transform value if present

Very often you need to apply some transformation on a value, but only if it's not`null` (avoiding `NullPointerException`):

```
if(x != null) {
    String t = x.trim();
    if(t.length() > 1) {
        print(t);
    }
}
```

This can be done in much more declarative way using `map()`:

```
opt.
    map(String::trim).
    filter(t -> t.length() > 1).
    ifPresent(this::print);
```

This becomes tricky. `Optional.map()` applies given function on a value inside`Optional` - but only if `Optional` is present. Otherwise nothing happens and`empty()` is returned. Remember that the transformation is type-safe - look at generics here:

```
Optional<String>  opt = //...
Optional<Integer> len = opt.map(String::length);
```

If `Optional<String>` is present `Optional<Integer> len` is present as well, wrapping length of a `String`. But if `opt` was empty, `map()` over it does nothing except changing generic type.

## `orElse()`/`orElseGet()` - turning empty `Optional<T>`to default `T`

At some point you may wish to unwrap `Optional` and get a hold of real value inside. But you can't do this if `Optional` is empty. Here is a pre-Java 8 way of handling such scenario:

```
intlen = (x != null)? x.length() : -1;
```

With `Optional` we can say:

```
intlen = opt.map(String::length).orElse(-1);
```

There is also a version that accepts `Supplier<T>` if computing default value is slow, expensive or has side-effects:

```
intlen = opt.
    map(String::length).
    orElseGet(() -> slowDefault());     //orElseGet(this::slowDefault)
```

## `flatMap()` - we need to go deeper

Imagine you have a function that does not accept `null` but may produce one:

```
publicString findSimilar(@NotNullString s) //...
```

Using it is a bit cumbersome:

```
String similarOrNull = x != null? findSimilar(x) : null;
```

With `Optional` it is a bit more straighforward:

```
Optional<String> similar = opt.map(this::findSimilar);
```

If the function we `map()` over returns `null`, the result of `map()` is an empty`Optional`. Otherwise it's the result of said function wrapped with (present)`Optional`. So far so good but why do we return `null`-able value if we have`Optional`?

```
publicOptional<String> tryFindSimilar(String s)  //...
```

Our intentions are clear but using `map()` fails to produce correct type. Instead we must use `flatMap()`:

```
Optional<Optional<String>> bad = opt.map(this::tryFindSimilar);
Optional<String> similar =       opt.flatMap(this::tryFindSimilar);
```

Do you see double `Optional<Optional<...>>`? Definitely not what we want. If you are mapping over a function that returns `Optional`, use `flatMap` instead. Here is a simplified implementation of this function:

```
public<U> Optional<U> flatMap(Function<T, Optional<U>> mapper) {
    if(!isPresent())
        returnempty();
    else{
        returnmapper.apply(value);
    }
}
```

## `orElseThrow()` - lazily throw exceptions on empty`Optional`

Often we would like to throw an exception if value is not available:

```
publiccharfirstChar(String s) {
    if(s != null&& !s.isEmpty())
        returns.charAt(0);
    else
        thrownewIllegalArgumentException();
}
```

This whole method can be replaced with the following idiom:

```
opt.
    filter(s -> !s.isEmpty()).
    map(s -> s.charAt(0)).
    orElseThrow(IllegalArgumentException::new);
```

We don't want to create an instance of exception in advance because creating an exception has significant cost.

## Bigger example

Imagine we have a `Person` with an `Address` that has a `validFrom` date. All of these can be `null`. We would like to know whether `validFrom` is set and in the past:

```
privatebooleanvalidAddress(NullPerson person) {
    if(person != null) {
        if(person.getAddress() != null) {
            finalInstant validFrom = person.getAddress().getValidFrom();
            returnvalidFrom != null&& validFrom.isBefore(now());
        } else
            returnfalse;
    } else
        returnfalse;
}
```

Quite ugly and defensive. Alternatively but still ugly:

```
returnperson != null&&
       person.getAddress() != null&&
       person.getAddress().getValidFrom() != null&&
       person.getAddress().getValidFrom().isBefore(now());
```

Now imagine all of these (`person`, `getAddress()`, `getValidFrom()`) are `Optional`s of appropriate types, clearly indicating they may not be set:

```
class Person {

    private final Optional<Address> address;

    public Optional<Address> getAddress() {
        return address;
    }

    //...
}

class Address {
    private final Optional<Instant> validFrom;

    public Optional<Instant> getValidFrom() {
        return validFrom;
    }

    //...
}
```

Suddenly the computation is much more streamlined:

```
return person.
        flatMap(Person::getAddress).
        flatMap(Address::getValidFrom).
        filter(x -> x.before(now())).
        isPresent();
```

Is it more readable? Hard to tell. But at least it's impossible to produce `NullPointerException` when `Optional` is used consistently.

## Converting `Optional<T>` to `List<T>`

I sometimes like to think about `Optional` as a collection[1] having either 0 or 1 elements. This may make understanding of `map()` and `flatMap()` easier. Unfortunately `Optional` doesn't have `toList()` method, but it's easy to implement one:

```
public static <T> List<T> toList(Optional<T> option) {
    return option.
            map(Collections::singletonList).
            orElse(Collections.emptyList());
}
```

Or less idiomatically:

```
public static <T> List<T> toList(Optional<T> option) {
    if (option.isPresent())
        return Collections.singletonList(option.get());
    else
        return Collections.emptyList();
}
```

But why limit ourselves to `List<T>`? What about `Set<T>` and other collections? Java 8 already abstracts creating arbitrary collection via `Collectors` API, introduced for `Streams`. The API is hideous but comprehensible:

```
public static <R, A, T> R collect(Optional<T> option, Collector<? super T, A, R> collector) {
    final A container = collector.supplier().get();
    option.ifPresent(v -> collector.accumulator().accept(container, v));
    return collector.finisher().apply(container);
}
```

We can now say:

```
import static java.util.stream.Collectors.*;

List<String> list = collect(opt, toList());
Set<String>  set  = collect(opt, toSet());
```

## Summary

`Optional<T>` is not nearly as powerful as `Option[T]` in Scala (but at least it doesn't allow wrapping `null`). The API is not as straightforward as `null`-handling and probably much slower. But the benefit of compile-time checking plus readability and

documentation value of `Optional` used consistently greatly outperforms disadvantages. Also it will probably replace nearly identical `com.google.common.base.Optional<T>` from Guava

1 - from theoretical point of view both *maybe* and *sequence* abstractions are *monads*, that's why they share some functionality

Posted 19th August 2013 by Tomasz Nurkiewicz
Labels: guava java 8 scala

0

View comments

1. **Anonymous** October 9, 2015 at 12:39 PM

   Greate art!
   Although you shouldn't use Optional(Address) as class field. See eg. BGoetz note http://stackoverflow.com/a/26328555

   Reply
   Replies
   1. **Pierre-Yves Saumont** November 17, 2015 at 6:19 PM

      Brian Goetz did not say so. He only said that "you probably should never use it for something that returns an array of results, or a list of results". And this is only because it is simpler to return an empty array or an empty list.

      Of course, if a property is optional, it should be an Optional and the corresponding getter should return an Optional. What else?

      Reply
2. **Anonymous** November 11, 2015 at 8:50 PM

   "There is also a version that accepts Supplier if computing default value is slow, expensive or has side-effects:" - I don't understand, how can orElseGet() help with optimisation?

   Reply
   Replies
   1. **Klitos** November 11, 2015 at 9:34 PM

      Because the slow method that computes the default will not be called unless necessary. If only orElse() was available, you would have to calculate the default value just to pass it to orElse() even if it will then be discarded.

      Reply
3. **Pierre-Yves Saumont** November 17, 2015 at 6:23 PM

   Hi,

   In the conversion to list example, why not using a method reference, replacing orElse(...) with .orElseGet(Collections::emptyList).

   Reply
   Replies
   1. **Tomasz Nurkiewicz** November 17, 2015 at 11:56 PM

      emptyList() returns a singleton so using orElseGet with constant value may actually be faster than Supplier.
   2. **Pierre-Yves Saumont** November 18, 2015 at 9:43 AM

      Yes, it is faster on the first call. However, orElseGet will not instantiate a Supplier. Using a lambda or a method reference will cause a (very small) overhead on the first call, when a method will be created to represent it. (And method reference is slightly faster, because it does not have to actually create a method, but can use a handle to the actual method instead.) So, if the Optional is empty, it will be slightly slower on the first call, and equivalent on subsequent calls. On the other hand, if the Optional is not empty, the Collections.emptyList() method will be called anyway, so what is supposed to be an optimization will in fact in most cases have worst performance. How much worst depends upon the proportion of calls with an empty Optional. If we suppose that there will be much more calls with a non empty one, the code using a method reference is definitely faster. It also protect your code against future modifications of the emptyList method implementation.