An alternative to Optional - nullable composition

Philipp Middendorf

2016-06-18 Sat

An alternative to optional - nullable composition

Prerequisite and disclaimer

In order to understand this post, you have to be familiar with **lambdas**, **default methods**, **functional interfaces** and ideally `Optional`. Also, please read this post more as a brain teaser than something to live by. I'm not advocating ditching `null`, `Optional` or using the functions below everywhere. It's an exercise in thinking functionally, above all.

Also, what I describe is probably some bastardized version of a simple concept that the Haskell people use on a regular basis. If so, please tell me, I'm eager to put all of this in context.

Prelude: Booooo! `null`!

Since day one, Java has had `null` as a special value that inhabits every type deriving from `Object`. In this article, I won't go into detail over why that's widely considered a bad idea. It's in the language an we have to deal with it properly. Typical code handling `null` values looks like this:

```
public void printMaleTenantsSpouse(Apartment a) {
 if(a != null) {
   Person t = a.getTenant();
   if(t != null) {
     if(t.isMale()) {
       Person spouse = t.getSpouse();
       if(spouse != null) {
         System.out.println(spouse.getName());
       }
     }
```

```
    }
  }
}
```

Here, I am assuming that *every* value in the object tree might be `null`, which mostly makes sense. An apartment doesn't have to have a tenant; that tenant doesn't have to have a spouse, and so on. Because we have to test for `null` on every level, this leads to some deeply nested code. I also threw in some other `if` just for kicks.

The savior, `Optional`!

In Java 8 though, there's `Optional`. This type and its operations were borrowed from languages like Haskell and C++, where you're in the opposite position: nothing can be null unless you wrap it in an optional type. It exposes the following operations (slightly simplified for readability):

- `Optional<T> empty()`: creates an empty `Optional`.
- `Optional<T> of(T t)`: creates a non-empty `Optional` from a non-null value (throws if `null` is passed).
- `Optional<T> ofNullable(T t)`: creates an empty `Optional` if passed `null`, otherwise a non-empty `Optional`.
- `T get()`: returns the value inside the optional, throwing if it's empty.
- `boolean isPresent()`: returns `true` if the optional is non-empty.
- `T orElse(T t)`: returns `t` if the `Optional` is empty, otherwise returns the value inside the `Optional`.
- `T orElseGet(Supplier<T> f)`: returns the result of applying `f` if the `Optional` is empty, otherwise returns the value inside the `Optional`.
- `T orElseThrow(Supplier<Throwable> f)`: throws if empty, otherwise returns the value inside the `Optional`.
- `void ifPresent(Consumer<T> f)`: executes `f`, passing the value if it's non-empty; otherwise does nothing.
- `Optional<T> filter(Predicate<T> p)`: checks the value inside the `Optional` against the predicate and maybe empties (literally) the optional accordingly.

- `Optional<U> map(Function<T,U> f)`: leaves empty `Optionals` empty, otherwise executes `f` on the value.

- `Optional<U> flatMap(Function<T,Optional<U»> f)`: leaves empty `Optionals` empty (switching the types), applies `f` to non-empty `Optionals` and returns the result.

Assuming we litter our data structures with getters that return `Optional` instead of `null` and using Java 8's method references, the code above could be written as follows:

```
public void printMaleTenantsSpouse(Optional<Apartment> a) {
  a.flatMap(Apartment::getTenant)
   .filter(Person::isMale)
   .flatMap(Person::getSpouse)
   .map(Person::getName)
   .ifPresent(System.out::println);
}
```

This is slightly shorter and looks cleaner, because the code doesn't expand to the right, and we don't have to assign any names which we only use once.

Great, now what's the problem? Well, we're actually misusing `Optional`! As `Optional`'s creator Stuart Marks often notes (for example in this StackOverflow answer), there are some "intended" uses for `Optional`, such as the return type of functions so you can "continue a chain of fluent method calls", which is actually one of the main reason why `Optional` was introduced – to make chained calls with `Stream` operations prettier:

```
collection.stream()
          .map(f)
          .filter(p)
          // findFirst returns the first found value in the stream
          // as an Optional<T> (the stream could be empty)
          .findFirst()
          .map(f2)
          .orElse(x);
```

However, using `Optional` as a parameter to a function (as seen above) or as a field of a data structure is "considered misuse", and many articles have been written about that fact (pro and contra). Also, `Optional` deliberately does not extend `Serializable` and framework support to serialize to JSON, for example, might have to to be enabled explicitly.

Also, from a more idealistic perspective, why have another type that represents the absence of a value? As we discovered, every variable can already be absent, containing `null`. What we're doing with `Optional` is wrapping it in another layer and unwrapping it at the end.

Burn `Optional`, hooray for `null`!

If we want this functional style operations, can't we define them on nullable types instead? Let's quickly go through the operations and how we might adapt them to work on plain types `T` that might contain `null`:

- `empty()`: we don't need a function for that, just write `null` and you're done!
- `of(t)`: this is also just packaging we don't need
- `ofNullable(t)`: see above
- `get()`: See above; just use the nullable value like you normally would; you'll get a `NullPointerException` if it's empty.
- `isPresent()`: this is just an if statement:

  ```
  public <T> boolean isPresent(T t) {
    return t != null;
  }
  ```

- `orElse(t)`: another if:

  ```
  public <T> boolean orElse(T t,T u) {
    return t != null ? t : Objects.requireNonNull(u);
  }
  ```

- `orElseGet(t)`: an if with a `get`:

  ```
  public <T> boolean orElseGet(T t,Supplier<T> f) {
    return t != null ? t : Objecs.requireNonNull(f.get());
  }
  ```

- `orElseThrow(t)`: an if with a throw:

```
public <T> boolean orElseThrow(T t,Supplier<Throwable> f) {
  if(t == null)
    throw f.get();
  return t;
}
```

- `ifPresent(f)`: slightly more interesting:

  ```
  public <T> void ifPresent(T t,Consumer<T> f) {
    if(t != null)
      f.accept(t);
  }
  ```

- `filter(p)`: given `null` (previously an empty `Optional`), just return `null`. Given non-null, return `null` if the predicate doesn't match, otherwise return the given value:

  ```
  public static <T> T filter(T t,Predicate<T> p) {
    return t == null || !p.test(t) ? null : t;
  }
  ```

- `map(f)`: does nothing if passed `null`, otherwise applies the function (which returns a `U`, not another `Optional`, so we have to be sure that it doesn't return `null`!)

  ```
  public static <T,U> T map(T t,Function<T,U> f) {
    if (t == null)
      return null;
    return Objects.requireNonNull(f.apply(t));
  }
  ```

- `flatMap(f)`: strikingly similar to `map`, but without the null check - the given function is allowed to return another `Optional` (or `null` in our case):

  ```
  public static <T,U> T flatMap(T t,Function<T,U> f) {
    if (t == null)
      return null;
    return f.apply(t);
  }
  ```

Using these operations, we can indeed rewrite the code as such:

```
ifPresent(
  map(
    flatMap(
      filter(
        flatMap(
          a,
          Apartment::getTenant),
        Person::isMale),
      Person::getSpouse),
    Person::getName),
  System.out::println)
```

Beautiful, isn't it! Such functional, very monadic!

So there's a reason Haskell has support for defining custom operators and an even more special syntax for `flatMap` operations, and Java has these chained method calls: functional code looks pretty ugly without them!

Burn `Optional` *and* `null`, use functions!

But we don't have to abandon ship just yet. What both the `Optional` code and the ugly-as-hell monster code above *did* hide is the conditional code, the "glue code" between our functions `Apartment::getTenant`, `Person::getName` and so on. Can't we concatenate these functions in a chained style, without using `Optional`, but hiding away the `if-else`?

What if instead of wrapping the *value*, we wrap the *function*? Instead of looking at a *value* that can be `null`, we're now looking at a *function* that can *return* `null`. In pseudocode with an annotation to remind you:

```
@FunctionalInterface
interface NullableFunction<T,U> {
  @Nullable
  U apply(@Nonnull T t);

  static <A,B> NullableFunction<A,B> of(NullableFunction<A,B> f) {
    return f;
  }
}
```

The function `of()` is necessary to force the type system to create the `NullableFunction` out of a method reference. I won't go into it here, so if you're not sure what it does, please read about `@FunctionalInterface`.

Now, assuming we have such a `NullableFunction`, we want to compose that with another function that returns `null`, thus defining the equivalent of `flatMap`:

```
interface NullableFunction<T,U> {
  // ...

  default <R> NullableFunction<T,R> flatMap(NullableFunction<U,R> f) {
    return t -> {
      U u = this.apply(t);
      if(u == null)
        return null;
      return f.apply(u);
    };
  }
}
```

Since we introduced a `null`-based `flatMap` before, this code is easy (we could even re-use the code from above, but I chose to expose it again). `filter` is a little tricky, because we have two choices: we can filter the argument of the function and the result (which can be null, of course). So I've provided both:

```
interface NullableFunction<T,U> {
  // ...
  default NullableFunction<T,U> filterArgument(Predicate<T> p) {
    return t -> {
      return p.test(t) ? this.apply(t) : null;
    };
  }

  default NullableFunction<T,U> filterResult(Predicate<U> p) {
    return t -> {
      U u = this.apply(t);
      if(u == null || !p.test(u))
        return null;
      return u;
```

```
    };
  }
}
```

The function `map`, is interesting, because it requires a function that does *not* return `null`. So we can't pass another `NullableFunction`. We could invent another interface `NonnullFunction`, but I've decided to just take `java.util.Function`:

```
interface NullableFunction<T,U> {
  // ...
  default NullableFunction<T,R> map(Function<U,R> p) {
    return t -> {
      U u = this.apply(t);
      if(u == null)
        return null;
      return p.apply(u);
    };
  }
}
```

The `orElse` family of functions can be defined, too:

```
interface NullableFunction<T,U> {
  // ...

  default NullableFunction<T,U> orElse(U fallback) {
    return t -> {
      U u = this.apply(t);
      return u == null ? fallback : u;
    };
  }

  default NullableFunction<T,U> orElseGet(Supplier<U> fallback) {
    return t -> {
      U u = this.apply(t);
      return u == null ? fallback.get() : u;
    };
  }
```

```
  // I am deliberately ignoring the fact that lambdas cannot
  // throw in Java.
  default NullableFunction<T,U> orElseThrow(Supplier<U> thrower) {
    return t -> {
      U u = this.apply(t);
      if(u == null)
        throw thrower.get();
      return u;
    };
  }
}
```

The function `ifPresent` can be defined. In `Optional`, it returns `void`, but it's much more usable as an equivalent of the `Stream` function `peek`, so you can continue chaining:

```
interface NullableFunction<T,U> {
  // ...

  default NullableFunction<T,U> ifPresent(Consumer<U> presenter) {
    return t -> {
      U u = this.apply(t);
      if(u != null)
        presenter.accept(u);
      return u;
    };
  }
}
```

There are some functions that cannot be transferred from `Optional`: `=get` doesn't make sense, because we have a function, not a value. There's nothing in it to get, it's the "between the values". `of` and `ofNullable` you cannot meaningfully define, because they refer to a single value, not a transformation. You *could* define `constant`, the function that, given any argument, ignores it and always returns a certain `t`:

```
interface NullableFunction<T,U> {
  // ...

  static NullableFunction<T,U> constant(U u) {
```

```
    return ignoreThisArgument -> {
      return u;
    };
  }
}
```

With this new machinery, let's rewrite the initial example again:

```
NullableFunction.of(Apartment::getTenant)
                .filter(Person::isMale)
                .flatMap(Person::getSpouse)
                .map(Person::getName)
                .ifPresent(System.out::println)
                .apply(a);
```

This looks as clean as the `Optional` solution, but doesn't extend to the right like the other solution without it. It is, however, not quite correct: We said that the `Apartment` we pass into it might be `null`, too. In the chaining methods we assumed that the value we pass is non-null (the *result*, however, can be null). To mitigate this, we add another convenience function `applyNullable` to `NullableFunction`:

```
interface NullableFunction<T,U> {
  // ...

  default U applyNullable(T t) {
    return t != null ? this.apply(t) : null;
  }
}
```

So the chain becomes:

```
NullableFunction.of(Apartment::getTenant)
                .filter(Person::isMale)
                .flatMap(Person::getSpouse)
                .map(Person::getName)
                .ifPresent(System.out::println)
                .applyNullable(a);
```

That's it folks. If you have questions or comments, please leave them in the according reddit thread in `/r/java`.