Optional...

What else?!

Optional. Een nieuw concept dat sinds Java 8 is geïntroduceerd. In zijn nog korte levensduur heeft Optional al voor veel opschudding gezorgd. De ene persoon vindt het een prachtige oplossing, de ander vindt het zo lelijk als de nacht. Echter, wat je mening ook is, het is onderdeel van de taal en je zal er hoe dan ook mee te maken krijgen. Ondanks dat er al veel meningen zijn, zijn niet alle meningen gegrond en weet ook niet iedereen wat nu eigenlijk het doel is van Optional. In dit artikel laat ik mijn licht schijnen op Optional. Hierbij worden een aantal valkuilen aangestipt, zodat deze ons niet meer verrassen.

Wat is een Optional?

Optional is op een bepaalde manier de Java implementatie van de Maybe monad. Nu moet je niet gelijk schrikken of afhaken vanwege het gevreesde 'M' woord. Eigenlijk is een monad helemaal niet zo afschrikwekkend en zeker niet de Maybe monad. Een monad is niks meer of minder dan het omsluiten van een type om een specifieke casus of gedrag te bedienen. In dit geval het feit dat het type de waarde *null* kan zijn met een mogelijke *NullPointerException* tot gevolg. Om het te simplificeren, kunnen we het eigenlijk beschouwen als een wrapper die een gebruiker dwingt om te kijken of de waarde wel of niet aanwezig is.

Voor wie niet zo bekend is met Optional hier in vogelvlucht de belangrijkste zaken. Stel dat we in Java een functie hebben die een User teruggeeft. De waarde van User kan null zijn wanneer deze bijvoorbeeld niet te vinden is. Hierdoor moet een gebruiker altijd eraan denken om een null check uit te voeren, voordat we iets aanroepen op de teruggeven User (zie **Listing 1)**.

In plaats daarvan kunnen we een Optional teruggeven. Nu kunnen we niet zomaar iets aanroepen op User, maar zijn we verplicht om eerst de Optional "uit te pakken" (zie **Listing 2**).

```
public User getUser() {
    return user;
}
```

Listing 1

```
public Optional<User> getUser() {
    return Optional.ofNullable(user);
}
```

Listing 2

```
Optional<String> a = Optional.of("Hello World");

Optional<String> b = Optional.empty();

Optional<String> c = null //doe dit niet !!!!!!
```

Listing 3

Door het teruggeven van Optional geef je het signaal richting de gebruiker van de functie dat de waarde mogelijk niet aanwezig is.

Logischerwijs zou je kunnen concluderen dat een Optional dus twee mogelijkheden heeft. Of hij is gevuld of hij is niet gevuld. Echter, we leven in een objectgeoriënteerde wereld. Dit betekent dat het mogelijk is om de waarde null toe te kennen aan Optional (zie **Listing 3**).

Als ik je vandaag maar één ding kan meegeven, dan is dat het volgende. Zorg ervoor dat je nooit, maar dan ook echt nooit, de waarde null toekent aan een Optional in je eigen code!

Brian Vermeer is Senior Software Engineer bij Blue4IT.

Ondanks dat Optional bedoeld is om de NullPointerException te voorkomen, kan het helaas dus nog steeds gebeuren. Het vergt dus nog steeds discipline van de ontwikkelaar om te zorgen dat het mechanisme werkt zoals het zou moeten werken.

Hoe om te gaan met optionals

Oké, leuk. Nu hebben we een Optional en dan? Om de daadwerkelijke waarde uit een Optional te halen, moeten we hem "uitpakken". Dit zou kunnen door simpelweg **`get()`** op de optional aan te roepen. Maar, als de Optional leeg is en we roepen **`get()`** aan, dan krijgen we te maken met een NoSuchElementException. Dit is ook niet iets wat we willen. Daarom zullen we eerst moeten kijken of de Optional überhaupt een waarde bevat voordat we **`get()`** kunnen aanroepen (zie **Listing 4**).

Het aanroepen van **`get()`** zonder te checken of de Optional gevuld is, kan dus leiden tot een Exception. Het zou verboden moeten worden! Daarom ben ik van mening dat de **`get()`** methode in z'n huidige vorm deprecated zou moeten zijn. Daarnaast is de imperatieve stijl van coderen, zoals hierboven, ook geen elegante oplossing. We moeten nu een **`isPresent()`** check uitvoeren waar we voorheen, voor het gebruik van Optional, een null check moesten uitvoeren. Wat mij betreft lood om oud ijzer en geen echte oplossing.

We kunnen Optional ook op een wat meer functionele of declaratieve manier gebruiken. Dit doen we door **`map()`** aan te roepen op de Optional. De lambda die we meegeven aan deze higher-order function zal alleen worden uitgevoerd als de Optional daadwerkelijk een waarde bevat (zie **Listing 5**).

Wanneer we **`execute()`** uitvoeren krijgen we, zoals verwacht, de output:

```
only run if optional is filled
```

Als we de waarde van **`maybeString`** veranderen naar **`Optional.empty()`** en **`execute()`** wordt wederom uitgevoerd, zien we vervolgens dat er geen output is in de console. Dit ziet er al een stuk interessanter en leesbaarder uit.

Alternatieve flows

Prima! Maar hoe moeten we dan omgaan met een alternatieve flow? Als we iets willen doen wanneer de Optional leeg is, dan hebben we drie keuzes:

- orElse()
- orElseGet()
- orElseThrow()

Met de laatste optie is het mogelijk om, met behulp van een supplier, een exceptie te gooien als de Optional leeg is. Dit is mogelijk in combinatie met **`map()`** maar het is niet per definitie nodig (zie **Listing 6**).

Wanneer we **`execute()`** aanroepen in bovenstaand voorbeeld, dan treed -zoals verwacht- een RuntimeException op.

Maar goed, we willen geen exceptie, maar daadwerkelijk een alternatieve functionaliteit toepassen wanneer de Optional geen

```
public String doSomething() {
   Optional<String> foo = Optional.empty();
   if (foo.isPresent()) {
       return foo.get();
   }
   return "";
}
```

Listing 4

```
public void execute() {
   Optional<String> maybeString = Optional.of("foo");
   maybeString.map(this::runIfExist);
}
private String runIfExist(String str) {
   System.out.println("only run if optional is filled ");
   return str;
}
```

Listing 5

```
public void execute() {
   Optional<String> maybeString = Optional.empty();
   maybeString
       .map(this::runIfExist)
       .orElseThrow(() -> new RuntimeException("Optional was empty"));
}
```

Listing 6

```
public void execute() {
   Optional<String> maybeString = Optional.of("foo");
   String newString = maybeString
           .map(this::runIfExist)
           .orElse(runIfEmpty());
   System.out.println(newString);
}
private String runIfExist(String str) {
   System.out.println("only run if optional is filled ");
   return str;
}
private String runIfEmpty() {
   System.out.println("only run if empty");
   return "empty";
}
```

Listing 7

waarde heeft. Onderstaand voorbeeld lijkt op het eerste gezicht vrij voor de hand liggend (zie **Listing 7**).

Als we de code simpelweg van boven naar beneden lezen, dan ontstaat de verwachting dat **`newString`** de waarde **`"foo"`** gaat krijgen. De Optional **`maybeString`** is immers gevuld, dus logischerwijs zou de **`map()`** zijn werk moeten doen. Wanneer we daadwerkelijk **`execute()`** runnen, dan krijgen we de volgende output in de console:

```
only run if optional is filled
only run if empty
foo
```

De verwachting dat **`newString`** de waarde **`"foo"`** zou krijgen, klopt nog steeds. Iets wat we hier nog meer zien, is dat de string **`"only run if empty"`** ook in de console output staat. Dit was iets dat je wellicht niet zou verwachten. Het betekent dus, dat ondanks dat de Optional een waarde heeft, de methode in de **`orElse()`** alsnog uit- gevoerd wordt.

We veranderen de methode **`execute()`** zodat we **`orElseGet()`** gebruiken in plaats van **`orElse()`**. Dit betekent dat we de method call naar **`runIfEmpty()`** ook zullen moeten veranderen naar een supplier (zie **Listing 8**).

Als we nu **`execute()`** aanroepen, zien we het volgende:

```
only run if optional is filled
foo
```

Nu zien we, zoals we in eerste instantie zouden verwachten, dat **`runIfEmpty()`** niet uitgevoerd is. Wat mij betreft, is dit niet echt intuïtief. Je zou eigenlijk verwachten dat dit het algemene gedrag zou zijn, ook bij de normale **`orElse()`**.

Conclusie

Als eerste hebben we gezien dat Optional de waarde **`null`** kan krijgen. Technisch gezien behouden we dus de mogelijk op een **`NullPointerException`**. Echter, met verstandig gebruik en daarmee dus nooit een Optional de waarde null toekennen, kan een hoop ellende voorkomen worden.

Daarnaast zien we dat als we **`orElse()`** gebruiken om alleen een alternatieve

```
public void execute() {
   Optional<String> maybeString = Optional.of("foo");
   String newString = maybeString
           .map(this::runIfExist)
           .orElseGet(() -> runIfEmpty());
   System.out.println(newString);
}
```

Listing 8

waarde toe te kennen wanneer een Optional leeg is, er geen enkel probleem is. Je moet er dan wel absoluut zeker van zijn dat de code in de **`orElse()`** geen enkel side effect heeft. Deze code wordt immers, ongeacht de waarde van de Optional, alsnog uitgevoerd. Over het algemeen is **`orElseGet()`** de veiligere en juiste optie, zeker als we daadwerkelijk logica willen uitvoeren.

Door de karakteristieken van het declaratief programmeren is de mix-up snel gemaakt. Daarnaast zie je het resultaat pas wanneer de code daadwerkelijk uitgevoerd is. Uiteraard zou je goede testcode moeten schrijven, maar het is nog beter om het verschil bij voorbaat te weten. Het verschil is subtiel, maar de impact kan enorm zijn wanneer je niet voorzichtig bent. ■

ZORG ERVOOR DAT JE NOOIT, MAAR DAN OOK ECHT NOOIT, DE WAARDE NULL TOEKENT AAN EEN OPTIONAL IN JE EIGEN CODE