Functionele lenzen voor Java developers

Menig functionele taal, zoals bijv. Haskell, heeft een *optics* library, waaronder zich allerlei leuke concepten vinden, zoals lenzen en prisma's. Zo kan een functionele lens gebruikt worden om diep geneste, immutable data structuren te updaten. Vanuit Haskell's oogpunt een bijna noodzaak, omdat standaard records en tuples en dergelijke *immutable by design* zijn, maar voor ons Javanen kan een lens ook van pas komen wanneer wíj in onze Java projecten met geneste, immutable data structuren werken. Een introductie *ter leering ende vermeack*.

In de Profunctor Optics Modular Data Accessors (1) paper wordt het volgende beschreven:

Data accessors allow one to read and write components of a data structure, such as the fields of a record, the variants of a union, or the elements of a container. These data accessors are collectively known as optics; they are fundamental to programs that manipulate complex data.

De meeste Javanen zijn bekend met data accessors, zoals getters en setters, om gegevens uit en in een Java object/structuur te krijgen, maar zoals de auteurs van bovengenoemde paper vervolgens opmerken: hoe zijn deze zinvol te combineren als je object onderdeel is van een groter, samengesteld object? Daarvoor kun je een zogenaamde lens gebruiken. Zonder te diep in te gaan op de theorie achter profunctors (of profunctor optics -- zoek maar op!) biedt een lens de mogelijkheid om in te zoomen en te focussen op een klein stukje van een groter geheel. En: je kunt dit stukje lezen en wijzigen.

Immutable datastructuren

Belangrijk om te begrijpen is dat developers in functionele talen, zoals Lisp, Erlang, Haskell, Clojure, gewend zijn aan immutability (onveranderlijkheid), een immutable (2) object of structuur is na het aanmaken niet meer te wijzigen.

In Haskell zijn bijvoorbeeld variabelen standaard immutable. Er kan een tuple of record gebruikt worden om enkele velden bij elkaar te groeperen, maar er is geen manier om de data te manipuleren zonder impliciet deze te kopiëren. Met andere woorden: waarden in een structuur worden nooit in-place bijgewerkt, maar een update-operatie levert altijd een nieuwe structuur.

Zie Listing 1 voor een (mutable) Java bean met traditionele getters/setters.

```
class MutableEngine {
  int warp;
  MutableEngine(int warp) {
     this.warp = warp;
  }
  int getWarp() {
     return warp;
  }
  void setWarp(int warp) {
     this.warp = warp;
  }
}
```

Listing 1.

Een bekende manier om in Java dit object immutable te maken, is om deze in de constructor te initialiseren en niet toe te staan dat vervolgens nog de interne waarden aangepast (kunnen) worden. Zie Listing 2.

```
class Engine {
  final int warp;

  Engine(int warp) {
     this.warp = warp;
  }

  Engine setWarp(int warp) {
     return new Engine(warp);
  }
}
```

Listing 2.

Ted Vinke is een enthousiaste en leergierige senior Java developer werkzaam bij First8 Conclusion.

Het aanpassen van deze `Engine` lijkt tegenstrijdig, maar zoals je in het voorbeeld kunt zien: er kan best een setter zijn, maar die verandert niet de huidige instantie. Ze retourneert een nieuwe!

Geneste immutable datastructuren

Stel, naast `Engine` maken we ook de volgende immutable classes `Propulsion` en `Ship` volgens dezelfde principes, zie Listing 3.

Een `Ship` heeft een `Propulsion`, welke een `Engine` heeft, welke een warp speed heeft, een aardig genest structuurtje.

Laten we een `Ship` maken, initieel met warp 2. Zie Listing 4.

Wat als we de snelheid willen aanpassen naar warp 3? Dat zouden we zonder na te denken op de volgende manier zoals in Listing 5 proberen.

Dat is jammer! Het aanpassen van de `Engine` heeft géén effect op `Propulsion` of `Ship`.

Het muteren van deze (geneste) instanties lukt niet op deze manier. Alle setters retourneren simpelweg een kopie, dus het updaten van de gehele structuur is...gedoe.

Wat als we de huidige warp snelheid willen verdubbelen? Zie Listing 6.

Hier wordt niemand vrolijk van...

```
class Propulsion {

  enum Type { SUBSPACE, TACHYON, GRAVITON }
  final Type type;
  final Engine engine;

  Propulsion(Type type, Engine engine) {
     this.type = type;
     this.engine = engine;
  }

  Propulsion setType(Type type) {
     return new Propulsion(type, this.engine);
  }

  Propulsion setEngine(Engine engine) {
     return new Propulsion(this.type, engine);
  }

  // getters
}

class Ship {
  final String name;
  final Propulsion propulsion;

  Ship(String name, Propulsion propulsion) {
     this.name = name;
     this.propulsion = propulsion;
  }

  Ship setName(String name) {
     return new Ship(name, this.propulsion);
  }

  Ship setPropulsion(Propulsion propulsion) {
     return new Ship(this.name, propulsion);
  }

  // getters
}
```

Listing 3.

```
var ship = new Ship(“Enterprise”, new Propulsion(SUBSPACE, new Engine(2)));
// Ship(name=Enterprise, propulsion=Propulsion(type=SUBSPACE, engine=Engine(warp=2)))
```

Listing 4.

```
ship.getPropulsion().getEngine().setWarp(3);
// Ship(name=Enterprise, propulsion=Propulsion(type=SUBSPACE, engine=Engine(warp=2)))
```

Listing 5.

```
ship.setPropulsion(
   ship.getPropulsion().setEngine(
       ship.getPropulsion().getEngine().setWarp(
           ship.getPropulsion().getEngine().getWarp() * 2)
   )
);
// Ship(name=Enterprise, propulsion=Propulsion(type=SUBSPACE, engine=Engine(warp=4)))
```

Listing 6.

Lenzen to the rescue

Ik gebruik λ van **https://palatable.github.io/lambda**, een bibliotheek van functionele patterns voor Java.

Laten we starten met een simpele lens. De lens focust zich op een kleiner gedeelte van een grotere container middels een combinatie van getter en setter.

We hebben een lens gemaakt van de grotere `Engine` naar het kleinere warp snelheid, of meer in idiomatische termen: de lens is nu de "getter en setter" voor de warp snelheid. Zie Listing 7.

Er zijn een paar functies voor interactie met lenzen, welke de λ-bibliotheek vernoemd heeft naar de tegenhangers in Haskell, waar de volgende combinators genoemd worden:

view
set
over

Met onze nieuwe lens kunnen we view gebruiken om een waarde te lezen, zie Listing 8.

Het mooie is dat je meerdere (compatibele) lenzen tezamen in een compositie kunt gebruiken, waarbij elke lens inzoomt op z'n eigen gedeelte van de gehele structuur. "Compatibel" hier betekent dat het "kleinere" gedeelte van een lens X het "grotere" gedeelte is van een andere lens Y.

Hoe ziet zo'n compositie eruit? Zie hiervoor Listing 9.

```
import com.jnape.palatable.lambda.optics.Lens;

var engineWarpLens = Lens.simpleLens(Engine::getWarp,
Engine::setWarp);
```

Listing 7.

```
import com.jnape.palatable.lambda.optics.functions.View;

Integer warp = View.view(engineWarpLens, ship.getPropulsion().
getEngine());
// 2
```

Listing 8.

```
var engineWarpLens = Lens.simpleLens(Engine::getWarp,
Engine::setWarp);
var propulsionEngineLens = Lens.simpleLens(Propulsion::getEngine,
Propulsion::setEngine);
var shipPropulsionLens = Lens.simpleLens(Ship::getPropulsion,
Ship::setPropulsion);

var composed = shipPropulsionLens
 .andThen(propulsionEngineLens)
 .andThen(engineWarpLens);

Integer warp = View.view(composed, ship);
// 2
```

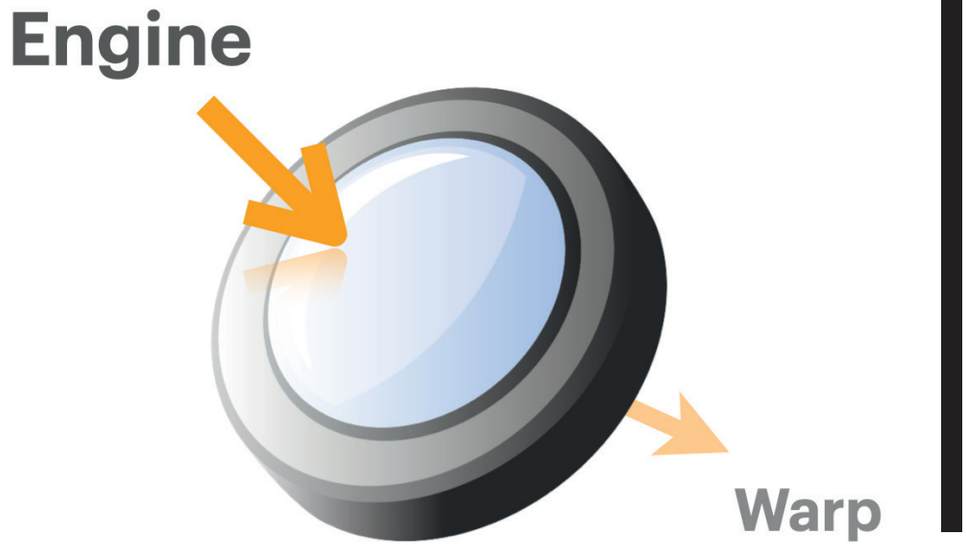
Listing 9.

```
import com.jnape.palatable.lambda.optics.functions.Set;

var newShip = Set.set(composed, 4, ship);
// Ship(name=Enterprise, propulsion=Propulsion(type=SUBSPACE,
engine=Engine(warp=4)))
```

Listing 10.

We kunnen nu set gebruiken om de warp snelheid op 4 te zetten -- en een compleet nieuw `Ship` terugkrijgen! Zie Listing 10.



Check nog eens de knullige code die we gebruikten om de warp snelheid te verdubbelen ;-) Met **over** kunnen we een functie meegeven die de huidige waarde (waar de lens naar wijst) teruggeeft voor ons om te manipuleren.

Laten we het nieuwe `Ship` met warp 4 van zojuist eens 2 keer zo hard gaan, zie Listing 11.

Conclusie

Een belangrijke reden voor het bestaan van lenzen in (talen zoals) Haskell, is dat het aanpassen van immutable datastructuren daar een stuk moeilijker is dan dat we gewend zijn in Java -- waar mutability de standaard is. Daarentegen moeten we in Java wel een stuk harder ons best doen om de immutability-voordelen te krijgen die de "functionalisten" out-of-the-box krijgen. Om onze Java objecten immutable te maken hebben we gelukkig keuze uit opties zoals Project Lombok's `@Value` (3) annotatie of Google's AutoValue (4).

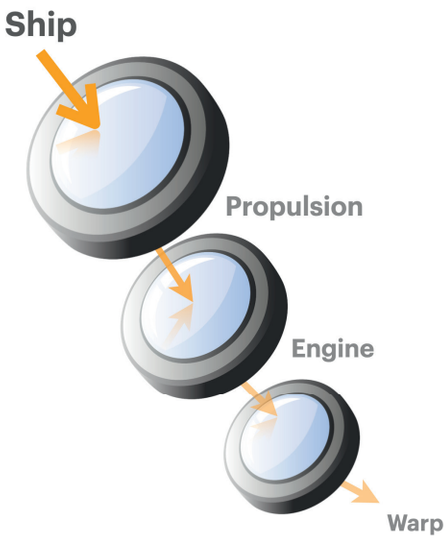
Door een lens te maken voor een specifiek gedeelte en deze the chainen met andere lenzen, is het best gemakkelijk om gedeelten van een diep geneste datastructuur//container te updaten.

Een nadeel kan zijn, dat - afhankelijk van de programmeertaal (of bibliotheek) - je voor elk veldje een lens moet definiëren. Met de hand. Ik heb lens generatoren gezien in Haskell en Scala die erg handig zijn, maar nog niet iets dergelijks in de nu gebruikte λ-bibliotheek of andere general-purpose alternatieven zoals Functional Java (5) . Alleen jLens (6) komt in de buurt met een annotation processor om lenzen te generen voor Java classes, maar die is al meer dan 8 jaar oud.

Een ander voordeel van lenzen zou kunnen zijn: encapsulatie -- de client van je lens weet niet (en hoeft niet te weten) wat de structuur is die erachter zit. In de mutable versie hieronder (of de immutable, knullige versie van eerder) overtreedt de client de Law of Demeter (7) en moet weten hoe de object graph afgelopen moet worden om helemaal bij de warp snelheid te komen:

```
ship.getPropulsion().getEngine().
setWarp(3);
```

Wanneer je iemand gewoon een lens geeft, weet diegene (of interesseert zich) niet hoe de lens gemaakt is. Diegene weet hoe omgegaan moet worden met de top-container, een `Ship`, en dat is genoeg, zie Listing 12.



Als laatste: een lens is z'n eigen type en kan (in principe) vrijelijk doorgegeven worden als argumenten aan methodes e.d.

Ik hoop dat deze introductie in functionele lenzen je fantasie geprikkeld heeft, en dat het wellicht een leuke toevoeging aan je Java-toolbox kan zijn wanneer je weer eens met immutable datastructuren aan de gang moet ;-) ■

```
import com.jnape.palatable.lambda.optics.functions.Over;

Over.over(composed, warp -> warp * 2, newShip);
// Ship(name=Enterprise, propulsion=Propulsion(type=SUBSPACE,
engine=Engine(warp=8)))
```

Listing 11.

```
var shipsWarpSpeedLens = // .. van ergens
Set.set(shipsWarpSpeedLens, 4, ship);
```

Listing 12.

LINKS

(1) **https://arxiv.org/ftp/arxiv/papers/1703/1703.10857.pdf**
(2) **https://tedvinke.wordpress.com/2018/06/15/functional-java-by-example-part-4-prefer-immutability/**
(3) **https://projectlombok.org/features/Value**
(4) **https://github.com/google/auto**
(5) **https://www.functionaljava.org/**
(6) **https://github.com/ppetr/jLens**
(7) **https://en.wikipedia.org/wiki/Law_of_Demeter**