NA EEN ZES JAAR LANG DUREND DEBAT HEBBEN DE LAMBDA'S NU EINDELIJK HUN WEG NAAR JAVA GEVONDEN

Lambda syntax opties

Lambda's kunnen in verschillende syntactische gedaante gebruikt worden. Het bovenstaande voorbeeld vertegenwoordigd de meest uitgebreide variant. Daarnaast kan het type van de input parameter van een lambda weggelaten worden:

```
new File("/").listFiles(f -> f.isDirectory());
```

Het mechanisme, die dat mogelijk maakt, wordt '*Type Inference*' genoemd. De compiler kent de signature van de enige method `(boolean accept(File f))` van `FileFilter` en kan daardoor de input type van de lambda achterhalen oftewel *interfereren*, waardoor deze niet expliciet genoemd hoeft worden. Het voordeel van *type inference* is voornamelijk code-reductie.
Als een lambda expressie niets anders doet dan een bestaande methode aanroepen, komen '*Method References*' van pas. Bovenstaand voorbeeld kan hiermee nog beknopter uitgedrukt worden:

```
new File("/").listFiles(File::isDirectory);
```

`File::isDirectory` is dus semantisch hetzelfde als `f -> f.isDirectory()`.
Method references komen in verschillende smaken. `File::isDirectory` is van het type '*Instance Method Reference*', omdat de implementatie van `listFiles de isDirectory()` methode op elke `File` aanroept, waarover heen gelopen wordt.
Instance method references zijn syntactisch compacter en (na wat gewenning) eenvoudiger leesbaar dan gewone lambda's. Dit is ook de voornaamste reden om ze te gebruiken.
Stel, je wilt met behulp van de `list–Files` methode bestanden filteren, die een symbolic link zijn. Hiervoor wil je gebruik maken van de static utility method boolean `isSymLink(File f)` van de commons FileUtils. *Static Method References* bieden hier uitkomst:

```
new File("/").listFiles(FileUtils::isSymLink);
```

Semantisch is `FileUtils::isSymLink` hetzelfde als `f -> FileUtils.isSymLink(f)`.
Static method references zijn ervoor gemaakt om functionaliteit van statische utility methode op een elegante manier in de context van een lambda te hergebruiken.

Functional interfaces

In Java 8 is de package `java.util.function` toegevoegd met een reeks nieuwe functional interfaces. Deze interfaces zijn in te delen in de volgende groepen:

`interface Function<T, R> { R apply(T t); }`
Een interface van het type `Function` definieert een operatie met een input parameter en als output van hetzelfde of een andere type als de input. De vertaling naar een lambda ziet er als volgt uit:

```
Function<Integer, Integer> square = in -> in * in
```

`interface Predicate<T> { boolean test(T t); }`
De Predicate interface definieert een operatie met een willekeurige input parameter en met een boolean als output, bijvoorbeeld:

```
Predicate<File> isDirectory = file -> file.isDirectory()
```

`interface Consumer<T> { void accept(T t); }`
Een Consumer definieert een operatie met een input parameter zonder resultaat, bijvoorbeeld:

```
Consumer<String> printMe = in -> System.out.println(in)
```

`interface Supplier<T> { T get(); }`
Een Supplier definieert een operatie zonder argumenten met als output een willekeurig type. Een voorbeeld hiervan is:

```
Supplier<LocalDate> now = () -> LocalDate.now();
```

De `java.util.function` package is al voorzien in een flink aantal variaties op bovenstaande interfaces. Zo is er voor elke primitieve variant een functional interface beschikbaar, zoals `IntPredicate`, `IntToLongFunction` etc. Functional interfaces met twee input parameters worden voorafgegaan door de prefix "Bi", zoals `BiPredicate` of `BiFunction`.
Deze nieuwe functional interfaces worden vooral toegepast in de vernieuwde Collections API van Java 8. Deze interfaces kunnen natuurlijk ook als input parameter(s) voor zelf gedefinieerde methoden gebruikt worden.
Een voorbeeld van het gebruik van een Function interface van het package `java.util.function` is te vinden in `java.util.Map`.