Default methoden maken multiple inheritance in Java 8 mogelijk

Met default methoden is zelfs een beperkte vorm van multiple inheritance mogelijk. Voor de komst van Java 8 was het alleen mogelijk om functionaliteit buiten een klasse via overerving van een enkele superklasse of via delegatie van een andere klasse te verkrijgen. Door default methoden kan functionaliteit van *verschillende* interfaces in een *enkele* klasse samengebracht worden. Daarnaast kan de klasse nog steeds functionaliteit van een superklasse overerven. Een hypothetisch voorbeeld:

```
abstract class Person {
    public Integer nextBirthdayInDays() {...}
}

interface RichComparable<T> extends
Comparable<T> {
    public default boolean greaterThan(T other) {
            return compareTo(other) > 0;
    }
//aanvullend: smallerThan, greaterThanEquals,
smallerThanEquals etc.
}

class Employee extends Person implements
RichComparable<Employee> {
    public int compareTo(Employee other) {
        return this.name.compareTo(other.name);
    }
}

Employee e1 = new Employee("Bakker",
"25-10-1981");
Employee e2 = new Employee("Jansen",
"12-12-1970");
e1.greaterThan(e2);
e1.nextBirthdayInDays();
```

Met bovenstaande constructie heeft de klasse Employee niet alleen beschikking over de functionaliteit van de superklasse Person, zoals bijvoorbeeld `nextBirthdayInDays`, maar ook van alle default methoden van de `RichComparable` interface, zoals `greaterThan`, `smallerThan` etc.
Omdat de logica van default methoden van meerdere interfaces in een klasse 'gemixt' kan worden en deze klasse bovendien logica van een superklasse kan overerven, beschikt Java 8 over een beperkte vorm van multiple inheritance. Hiermee kunnen aanzienlijk flexibeler API's gemaakt worden, die een API-ontwerper niet meer beperken tot het onderbrengen van functionaliteit in abstracte klasses alleen.
Een belangrijke kanttekening is echter wel dat multiple inheritance met default methoden beperkt is tot gedrag. Interfaces kunnen namelijk geen state bevatten. Als een API gemaakt wordt, waar state een belangrijke rol speelt, biedt alleen single inheritance (oftewel klasse inheritance) uitkomst.

The best of both worlds

Lambda's en default methoden bieden ongekende mogelijkheden voor hergebruik, flexibiliteit en code-reductie, waarmee krachtige API's geschreven kunnen worden. Ter afronding een voorbeeld over de in dit artikel beschreven concepten. Stel, je krijgt de opdracht persoonsgegevens uit een CSV bestand te filteren op basis van verschillende criteria. Gebruikmakend van de nieuwe mogelijkheden zou dit probleem met de volgende paar regels code opgelost kunnen worden:

```
class PersonParser {
     public static Person parse(String csvLine) {...}
}

List<Person> filterPersons(URL url, Predicate<Person> filter) {
    try {
    BufferedReader br = new BufferedReader(
      new InputStreamReader(url.openStream()))
        return br.lines()
                 .map(PersonParser::parse)
                 .filter(filter)
                 .collect(Collectors.toList());
  } catch (IOException e) {
    throw new RuntimeException(e);
  }
}
```

Sinds Java 8 is de BufferedReader voorzien van een `lines()` methode, die een `java.util.Stream` teruggeeft. Een `Stream` is een soort tijdelijke collectie, die een grote hoeveelheid krachtige methoden bevat, zoals map, filter etc., die lambda's als input parameter verwachten. Met onder andere de aanroep naar de filter methode op de stream, worden alleen die personen geselecteerd die voldoen aan het opgegeven Predicate. Met deze opzet zijn we als gebruiker van deze API ongekend flexibel. Met behulp van lambda's kunnen wij filter expressies op beknopte wijze bepalen om zo de lijst van personen op verschillende manieren te filteren:

```
filterPersons(csvUrl, person -> person.getAge() > 30)
filterPersons(csvUrl, person -> person.startsWith("A"))
```

Zonder lambda's en default methoden zou de `filterPersons` functionaliteit ongeveer twee keer zoveel code bevatten en bovendien aanzienlijk minder gebruiksvriendelijk zijn. Dit geeft aan dat we met Java 8 een mooie toekomst tegemoet gaan, waarbij we met minder aanzienlijk meer voor elkaar kunnen krijgen. ■