Aan een Map is in Java 8 de volgende handige methode toegevoegd:

```
V merge(K key, V value,
          BiFunction<? super V, ?
super V, ? extends V> remappingFunction)
```

Aan de BiFunction kunnen we met een lambda de volgende implementatie geven:

```
BiFunction<Integer, Integer, Integer>
maxFun1 =
  (oldValue, newValue) ->
       Math.max(oldValue, newValue);
```

Of nog beknopter, gebruik makende van een static method reference:

```
BiFunction<Integer, Integer, Integer>
maxFun2 = Math::max;
```

Als de merge methode wordt aanroepen, zal de expliciete verwijzing naar `BiFunction` over het algemeen niet gebruikt worden. In plaats daarvan zal de lambda expressie, die hoort bij de `BiFunction` inline, gebruikt worden:

```
Map<String, String> map = new HashMap<>();
map.put("java", 7);
map.merge("java", 8, Math::max);
```

Na het uitvoeren van de merge levert dit de waarde 8.
Het is dus belangrijk om te weten hoe een functional interface vertaald wordt naar een lambda. In het begin zal dat wat tijd nodig hebben, omdat de javadoc bij een methode, die een functional interface als argument gebruikt, slechts het type van deze interface vermeld. De input en output parameters, van de door de functional interface gedefinieerde methode, zijn niet direct af te leiden. Dit is echter juist noodzakelijk voor de vertaling naar een lambda expressie.

Default methoden

In Java 8 is het mogelijk om methodes van een interface te voorzien van een concrete implementatie. Voor de betreffende methode dient hiervoor het nieuwe keyword '*default*' gedeclareerd te worden, gevolgd door een implementatie. Als een klasse de interface implementeert, hoeft er voor de default methoden van de interface geen implementatie geleverd te worden. Het is echter wel mogelijk om de default methode te overschrijven.
Met default methoden is het mogelijk om nieuwe methoden aan bestaande interfaces toe te voegen, zonder hierbij bestaande implementaties te breken. In die zin kun je een interface uitbreiden met behoud van backward compatibility.
In de vernieuwde Collections API zijn veel voorbeelden van default methoden te vinden, zoals onder andere in de `java.util.Collection` interface:

```
default boolean removeIf(Predicate<? super E> filter) {
  Objects.requireNonNull(filter);
  boolean removed = false;
      final Iterator<E> each = iterator();
      while (each.hasNext()) {
           if (filter.test(each.next())) {
                each.remove();
                removed = true;
           }
      }
      return removed;
}
```

Zou je als programmeur in een versie voor Java 8 zelf een implementatie van `java.util.Collection` hebben geschreven, dan is die nog steeds op een Java 8 JVM te gebruiken. Zelfs als de implementatie geen `removeIf` methode bevat. In plaats daarvan wordt de default implementatie van `removeIf` gebruikt.
Door de toepassing van default methoden is het nu mogelijk om `myPeopleList.sort(...)` te schrijven in plaats van `Collections.sort(myPeopleList, ...)`. Dit kan omdat aan de `java.util.List` interface een default methode `sort(...)` is toegevoegd, waarvan de implementatie de methode van `Collections.sort(...)` aanroept.
Klassen die interfaces met default methoden implementeren, krijgen rijkere functionaliteit en zijn in het gebruik meer objectgeoriënteerd. Een ander voorbeeld vinden we in de Predicate interface, die is voorzien van een aantal default methoden, bijvoorbeeld:

WE GAAN MET JAVA 8 EEN MOOIE TOEKOMST TEGEMOET

```
default Predicate<T> and(Predicate<? super T> other) {
     return (t) -> test(t) && other.test(t);
}
```

We kunnen nu dus direct de volgende compositie toepassen zonder een extra helper klasse te moeten schrijven:

```
Predicate<Person> lastNameStartsWithA =
        person -> person.startsWith("A")
Predicate<Person> ageAbove30 =
        person -> person.getAge() > 30
Predicate<Person> composed =
lastNameStartsWithA.and(ageAbove30);
```