HO GENT

Functioneel programmeren

streams

Table of Contents

1. Doelstellingen

- Leren wat functioneel programmeren inhoudt en hoe het object-georiënteerd programmeren aanvult.
- Functioneel programmeren gebruiken om bepaalde programmeertaken te vereenvoudigen.
- Lambda expressies schrijven die functionele interfaces implementeren.
- Wat zijn streams? Hoe worden stream pipelines gevormd uit streambronnen, intermediate operaties en terminal operaties.
- Uitvoeren van operaties op IntStreams, zoals forEach, count, min, max, sum, average, reduce, filter en sorted.
- Uitvoeren van operaties op Streams, zoals filter,map, sorted, collect, forEach, findFirst, distinct, mapToDouble en reduce.

2. Inleiding

- Tot Java SE 8 ondersteunde Java drie programmeerparadigma's:
  - procedureel programmeren,
  - object-geörienteerd programmeren en
  - generiek programmeren.
    ⇒ Java SE 8 nu ook **functioneel** programmeren.
- Project Lambda: http://openjdk.java.net/projects/lambda
- Tot nu toe: je specifieert **hoe** een taak moet worden uitgevoerd.

```
int sum = 0, values[] = {1,2,3,4,5};
for (int counter = 0; counter < values.length; counter++)
    sum += values[counter];
```

- **External** iteratie
  - Gebruik van een lus om te itereren over een collectie van elementen.
  - Vereist sequentiële benadering van de elementen.
  - Vereist veranderlijke variabelen (sum en counter).
- Functioneel programmeren
  - Specifieer **wat** je wil in een taak, maar niet hoe.
- **Internal** iteratie
  - Laat de bibliotheek de manier bepalen om over een collectie van elementen te itereren.
  - Internal iteratie is gemakkelijker voor parallelle uitvoering.
- Functioneel programmeren legt de klemtoon op immutability, het niet aanpassen van de aangesproken databron.

3. Functionele interfaces

- Functionele interfaces, ook gekend als single abstract method (SAM) interfaces (bevatten één abstracte methode).
  - Voorbeeld: EventHandler

```
@FunctionalInterface
public interface EventHandler<T extends Event>
extends EventListener
```

Handler for events of a specific class / type.

Since:

`JavaFX 2.0`

Method Summary

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method and Description
`void`	**`handle`**`(T event)` Invoked when a specific event of the type for which this handler is registered happens.

- Belangrijke toegevoegde functionele interfaces (sedert JDK8) om functioneel te programmeren in Java = werken met streams en lambda's (zie verder).

	Methode	Voorbeeld
Predicate<T>	boolean test(T t)	Voorwaarde controleren
Consumer<T> en ook BiConsumer<T,U>	void accept(T t)	Afdrukken van een waarde
Function<T,R>	R apply(T t)	Geef de naam van een artiest
Supplier<T>	T get()	Ophalen waarde
UnaryOperator<T>	T apply(T t)	Logische not
BinaryOperator<T> En ook BiFunction<T,U,R>	T apply(T t1, T t2)	2 getallen vermenigvuldigen

4. Lambda expressies

- Lambda expressie
  - anonieme methode
  - snelschrift notatie voor het implementeren van een functionele interface.
- Kan gebruikt worden waar functionele interfaces worden verwacht.
- Een lambda expressie bestaat uit een parameterlijst gevolgd door een pijltoken en een body:

```
(parameterList) -> {statements}
```

- Vb: lambda ontvangt twee ints en geeft hun som terug:

```
(int x, int y) -> {return x + y;}
```

Deze lambda's body is een blok dat één of meerdere statements kan bevatten tussen accolades.
Er zijn meerdere variaties mogelijk:

```
(x, y) -> {return x + y;}
```

```
(x, y) -> x + y
```

- Bestaat de parameterlijst uit één parameter, dan mogen de haakjes weg:

```
value -> System.out.printf("%d ", value)
```

- Een lambda met lege parameterlijst:

```
( ) -> System.out.println("Welcome to lambdas!")
```

Voorbeeld gebruik van een lambda:

```
button.setOnAction(
        new EventHandler<ActionEvent>()
            {
                @Override
                public void handle(ActionEvent event)
                {
                    System.out.println("button clicked");
                }
            });

button.setOnAction(
            event -> System.out.println("button clicked"));
```

- extra voorbeelden:

```
        String[] friendsArray = {"Brian","Nate","Neal","Sara","Betty"};
        List<String> friends = new ArrayList<>(Arrays.asList(friendsArray));

        for(String name : friends)
            System.out.println(name);
        System.out.println();

        friends.forEach(new Consumer<String>() {
            @Override
            public void accept(String t)
            {
                System.out.println(t);
            }
        });
        System.out.println();
```

```
1         friends.forEach((String t) -> { System.out.println(t);});
2         System.out.println();
3
4         friends.forEach( t -> System.out.println(t));
5         System.out.println();
6
7         friends.forEach( System.out::println);
8         System.out.println();
```

5. Streams - algemeen

- Streams zijn objecten van
    - klassen die de interface Stream (package java.util.stream) implementeren

    - Één van de gespecialiseerde stream interfaces voor verwerking van int, long of double waarden
- Stream pipelines
    - Laat elementen een reeks van verwerkingsstappen doorlopen.
    - Pipeline
        - begint met een databron,
        - voert meerdere intermediate operaties uit op de elementen van de databron en
        - eindigt met een terminal operatie.
    - Wordt gevormd door geketende methode aanroepen.
- Streams bewaren geen data
    - Eenmaal een stream is uitgevoerd kan het niet worden herbruikt, omdat het geen kopij bijhoudt van de originele databron.
- Intermediate (=**tussentijdse**) operatie
    - specifieert een taak op elementen van een stream en resulteert altijd in een nieuwe stream.
    - Zijn lazy: worden pas uitgevoerd als een terminal operatie wordt aangeroepen.
- Terminal (=**eind**) operatie
    - Start de verwerking van de stream pipeline's intermediate operaties
    - Creëert een resultaat
    - Zijn eager: voeren de gevraagde operatie uit wanneer ze worden aangeroepen.
- Samengevat:

Volgorde:
starten bij een databron(=datatype) → plaatsen van een Stream → eindigen bij resultaat(=datatype)

- Plaatsen van een Stream op de databron:
    - Stream wijzigt databron **NIET**
    - Stream is **GEEN** datatype
    - Is ontwikkeld voor lambda's
    - **GEEN** geïndexeerde toegang
- intermediate operaties = bewerkingen op de databron
- terminal operaties ⇒ resulteren terug in een datatype (Vb: List)

6. Creatie van een stream en stream operaties

6.1. IntStream operaties

- De gebruikte technieken gelden ook voor LongStreams and DoubleStreams voor respectievelijk long en double waarden.

```
1  public class IntStreamOperations
2  {
3     public static void main(String[] args)
4     {
5        int[] values = {3, 10, 6, 1, 4, 8, 2, 5, 9, 7};
6
7        // display original values
8        System.out.print("Original values: ");
9        IntStream.of(values)
10                .forEach(value -> System.out.printf("%d ", value)); ①
11       System.out.println();
12
13       // count, min, max, sum and average of the values
14       System.out.printf("%nCount: %d%n", IntStream.of(values).count());
15       System.out.printf("Min: %d%n",
16          IntStream.of(values).min().getAsInt());
17       System.out.printf("Max: %d%n",
18          IntStream.of(values).max().getAsInt());
19       System.out.printf("Sum: %d%n", IntStream.of(values).sum());
20       System.out.printf("Average: %.2f%n",
21          IntStream.of(values).average().getAsDouble()); ②
22
23       // sum of values with reduce method
24       System.out.printf("%nSum via reduce method: %d%n",
25          IntStream.of(values)
26                   .reduce(0, (x, y) -> x + y));
27
28       // sum of squares of values with reduce method
29       System.out.printf("Sum of squares via reduce method: %d%n",
30          IntStream.of(values)
31                   .reduce(0, (x, y) -> x + y * y));
32
33       // product of values with reduce method
34       System.out.printf("Product via reduce method: %d%n",
35          IntStream.of(values)
36                   .reduce(1, (x, y) -> x * y)); ③
37
38       // even values displayed in sorted order
39       System.out.printf("%nEven values displayed in sorted order: ");
40       IntStream.of(values)
```

```
41                .filter(value -> value % 2 == 0) ④
42                .sorted()    ⑤
43                .forEach(value -> System.out.printf("%d ", value));
44        System.out.println();
45
46        // odd values multiplied by 10 and displayed in sorted order
47        System.out.printf(
48           "Odd values multiplied by 10 displayed in sorted order: ");
49        IntStream.of(values)
50                .filter(value -> value % 2 != 0)
51                .map(value -> value * 10)  ⑥
52                .sorted()
53                .forEach(value -> System.out.printf("%d ", value));
54        System.out.println();
55
56        // sum range of integers from 1 to 10, exclusive
57        System.out.printf("%nSum of integers from 1 to 9: %d%n",
58           IntStream.range(1, 10).sum()); ⑦
59
60        // sum range of integers from 1 to 10, inclusive
61        System.out.printf("Sum of integers from 1 to 10: %d%n",
62           IntStream.rangeClosed(1, 10).sum()); ⑧
63     }
64 } // end class IntStreamOperations
```

① IntStream static methode **of** krijgt een int array als argument en geeft een IntStream terug om de waarden in de array te verwerken.
IntStream methode **forEach** (terminal operatie) krijgt als argument een object dat de IntConsumer functional interface (package java.util.function) implementeert. Deze interface **accept** methode krijgt één int waarde en voert er een taak mee uit.
De **forEach** loopt over de stream en voert voor elk element dezelfde functie uit, hier afdrukken van het element.

② Klasse IntStream voorziet:

- count geeft aantal elementen terug
- min geeft de kleinste int terug
- max geeft de grootste int terug
- sum geeft de som van alle ints terug
- average geeft een OptionalDouble (package java.util) terug, die bevat het gemiddelde van de ints als een waarde van het type double.
  Klasse OptionalDouble's **getAsDouble** methode geeft de double in het object terug of gooit een NoSuchElementException. Om deze exception te voorkomen, kan je de methode **orElse** gebruiken. Deze geeft de OptionalDouble's waarde terug als er een is, of de waarde die je doorgeeft aan orElse (zie ook Extra's).

③ Je kan je eigen verkortingen definiëren voor een IntStream door zijn **reduce** methode aan te roepen.

- Eerste argument is een waarde die gebruikt wordt als begin van de reduction operatie

- Tweede argument is een object dat de IntBinaryOperator functional interface implementeert.
  Het eerste argument van de methode reduce wordt een identity waarde genoemd. Als deze waarde gecombineerd wordt met een stream element, gebruikmakend van IntBinaryOperator , dan levert dat de originele waarde van dat element op.

④ Filteren van elementen volgens een bepaalde voorwaarde.

- IntStream methode **filter** ontvangt een object dat de IntPredicate functional interface (package java.util.function) implementeert.

⑤ IntStream methode **sorted** ordert de elementen van de stream in oplopende volgorde(by default).

⑥ **Mapping** is een intermediate operatie die de elementen van een stream omzet naar nieuwe waarden en een nieuwe stream met de resultaten creëert.
De nieuwe stream kan ook van een ander type zijn.

⑦ IntStream methoden **range** en **rangeClosed** produceren elk een geordende reeks van int waarden. Beide methoden hebben twee int argumenten die het bereik van de waarden voorstellen.

- Methode **range** produceert een reeks van waarden vanaf het eerste argument tot, niet inbegrepen, het tweede argument.

⑧ Methode **rangeClosed** produceert een reeks van waarden, beide argumenten inbegrepen.

6.2. Arrays en streams

```java
public class ArraysAndStreams
{
   public static void main(String[] args)
   {
      Integer[] values = {2, 9, 5, 0, 3, 7, 1, 4, 8, 6};

      // display original values
      System.out.printf("Original values: %s%n", Arrays.asList(values));

      // sort values in ascending order with streams
      System.out.printf("Sorted values: %s%n",
         Arrays.stream(values)      ①
               .sorted()
               .collect(Collectors.toList())); ②

      // values greater than 4
      List<Integer> greaterThan4 =
         Arrays.stream(values)
               .filter(value -> value > 4)
               .collect(Collectors.toList());
      System.out.printf("Values greater than 4: %s%n", greaterThan4);

      // filter values greater than 4 then sort the results
      System.out.printf("Sorted values greater than 4: %s%n",
         Arrays.stream(values)
               .filter(value -> value > 4)
               .sorted()
               .collect(Collectors.toList()));

      // greaterThan4 List sorted with streams
      System.out.printf(
         "Values greater than 4 (ascending with streams): %s%n",
         greaterThan4.stream()
               .sorted()
               .collect(Collectors.toList()));
   }
} // end class ArraysAndStreams
```

① Klasse Arrays biedt overloaded **stream** methoden voor het creëren van IntStreams, LongStreams en DoubleStreams uit int, long en double arrays of reeksen van elementen uit de arrays.

② Voor het creëren van een verzameling die de resultaten van een stream pipeline bevat, kan je de Stream methode **collect** (terminal operatie) gebruiken.
Methode **collect** met één argument krijgt een object dat de interface Collector (package java.util.stream) implementeert, die specifieert hoe de veranderlijke reductie moet worden uitgevoerd.

- Collectors methode **toList** zet een Stream<T> om in een List<T> collectie.

```java
public class ArraysAndStreams2
{
   public static void main(String[] args)
   {
      String[] strings =
         {"Red", "orange", "Yellow", "green", "Blue", "indigo", "Violet"};

      // display original strings
      System.out.printf("Original strings: %s%n", Arrays.asList(strings));

      // strings in uppercase
      System.out.printf("strings in uppercase: %s%n",
         Arrays.stream(strings)
               .map(String::toUpperCase)  ①
               .collect(Collectors.toList()));

      // strings greater than "m" (case insensitive) sorted ascending
      System.out.printf("strings greater than m sorted ascending: %s%n",
         Arrays.stream(strings)
               .filter(s -> s.compareToIgnoreCase("m") > 0)
               .sorted(String.CASE_INSENSITIVE_ORDER) ②
               .collect(Collectors.toList()));

      // strings greater than "m" (case insensitive) sorted descending
      System.out.printf("strings greater than m sorted descending: %s%n",
         Arrays.stream(strings)
               .filter(s -> s.compareToIgnoreCase("m") > 0)
               .sorted(String.CASE_INSENSITIVE_ORDER.reversed()) ③
               .collect(Collectors.toList()));
   }
} // end class ArraysAndStreams2
```

① Stream methode **map** zet ieder element om naar een nieuwe waarde en maakt een nieuwe stream met hetzelfde aantal elementen als de originele stream.

- Een methodereferentie is een snelschrift notatie voor een lambda expressie.
- ClassName::instanceMethodName stelt een methodereferentie voor van een instantiemethode van de klasse.
- objectName::instanceMethodName stelt een methodereferentie voor, voor een instantiemethode die aangeroepen wordt op een specifiek object.
- ClassName::staticMethodName stelt een methodereferentie voor, voor een static methode van een klasse.

② Stream methode **sorted** kan een Comparator als argument ontvangen, om zo de sorteervolgorde vast te leggen.

- By default, methode sorted gebruikt de natuurlijke volgorde voor de stream's element type.
- Voor Strings, de natuurlijke volgorde is case sensitive, dit betekent dat "Z" is kleiner dan "a".

③ Functional interface Comparator's default methode **reversed** keert een bestaande Comparator's volgorde om.

6.3. Lists en streams

```
1  public class ProcessingEmployees
2  {
3     public static void main(String[] args)
4     {
5        // initialize array of Employees
6        Employee[] employees = {
7           new Employee("Jason", "Red", 5000, "IT"),
8           new Employee("Ashley", "Green", 7600, "IT"),
9           new Employee("Matthew", "Indigo", 3587.5, "Sales"),
10          new Employee("James", "Indigo", 4700.77, "Marketing"),
11          new Employee("Luke", "Indigo", 6200, "IT"),
12          new Employee("Jason", "Blue", 3200, "Sales"),
13          new Employee("Wendy", "Brown", 4236.4, "Marketing")};
14
15       // get List view of the Employees
16       List<Employee> list = Arrays.asList(employees);
17
18       // display all Employees
19       System.out.println("Complete Employee list:");
20       list.stream().forEach(System.out::println);  ①
21
22       // Predicate that returns true for salaries in the range $4000-$6000
23       Predicate<Employee> fourToSixThousand =
24          e -> (e.getSalary() >= 4000 && e.getSalary() <= 6000);  ②
25
26       // Display Employees with salaries in the range $4000-$6000
27       // sorted into ascending order by salary
28       System.out.printf(
29          "%nEmployees earning $4000-$6000 per month sorted by salary:%n");
30       list.stream()
31           .filter(fourToSixThousand)
32           .sorted(Comparator.comparing(Employee::getSalary)) ③
33           .forEach(System.out::println);
34
35       // Display first Employee with salary in the range $4000-$6000
36       System.out.printf("%nFirst employee who earns $4000-$6000:%n%s%n",
37          list.stream()
38              .filter(fourToSixThousand)
39              .findFirst()
40              .get());  ④
41
42       // Functions for getting first and last names from an Employee
43       Function<Employee, String> byFirstName = Employee::getFirstName;
44       Function<Employee, String> byLastName = Employee::getLastName;
45
```

```
46       // Comparator for comparing Employees by first name then last name
47       Comparator<Employee> lastThenFirst =
48          Comparator.comparing(byLastName).thenComparing(byFirstName);  ⑤
49
50       // sort employees by last name, then first name
51       System.out.printf(
52          "%nEmployees in ascending order by last name then first:%n");
53       list.stream()
54           .sorted(lastThenFirst)
55           .forEach(System.out::println);
56
57       // sort employees in descending order by last name, then first name
58       System.out.printf(
59          "%nEmployees in descending order by last name then first:%n");
60       list.stream()
61           .sorted(lastThenFirst.reversed())
62           .forEach(System.out::println);
63
64       // display unique employee last names sorted
65       System.out.printf("%nUnique employee last names:%n");
66       list.stream()
67           .map(Employee::getLastName)
68           .distinct()    ⑥
69           .sorted()
70           .forEach(System.out::println);
71
72       // display only first and last names
73       System.out.printf(
74          "%nEmployee names in order by last name then first name:%n");
75       list.stream()
76           .sorted(lastThenFirst)
77           .map(Employee::getName)
78           .forEach(System.out::println);
79
80
81       // sum of Employee salaries with DoubleStream sum method
82       System.out.printf(
83          "%nSum of Employees' salaries (via sum method): %.2f%n",
84          list.stream()
85              .mapToDouble(Employee::getSalary)   ⑦
86              .sum());
87
88       // calculate sum of Employee salaries with Stream reduce method
89       System.out.printf(
90          "Sum of Employees' salaries (via reduce method): %.2f%n",
91          list.stream()
92              .mapToDouble(Employee::getSalary)
93              .reduce(0, (value1, value2) -> value1 + value2));
94
95       // average of Employee salaries with DoubleStream average method
96       System.out.printf("Average of Employees' salaries: %.2f%n",
```

```
97          list.stream()
98              .mapToDouble(Employee::getSalary)
99              .average()
100             .getAsDouble());
101    } // end main
102 } // end class ProcessingEmployees
```

① Instantiemethode referentie **System.out::println** wordt doorgegeven naar de Stream methode **forEach**. Deze wordt door de compiler omgezet naar een object dat de Consumer functional interface implementeert.

② Om een lambda te hergebruiken, kan je het toekennen aan een variabele van het juiste functional interface type.

③ De Comparator interface's static methode **comparing** ontvangt een Function dat gebruikt wordt om een waarde uit een object in de stream te halen, dat verder gebruikt wordt in vergelijkingen en een Comparator object teruggeeft.

④ Stream methode **findFirst** is een short-circuit terminal operation die de stream pipeline uitvoert en stopt met de verwerking vanaf het moment dat het eerste object in de stream pipeline is gevonden.

⑤ Om objecten te sorteren op twee velden, creëer je een Comparator die twee Functions gebruikt.

- Eerst roep je de Comparator methode **comparing** op om een Comparator met een eerste Function te creëren.
- Op de resulterende Comparator, roep je de methode **thenComparing** met de tweede Function.
  De resulterende Comparator vergelijkt de objecten volgens de eerste Function en, voor objecten die gelijk zijn, vervolgens op de tweede Function.

⑥ Stream methode **distinct** verwijdert dubbele objecten uit de stream.

⑦ Stream methode mapToDouble beeldt objecten af op double waarden en geeft een DoubleStream terug.

- Stream methode **mapToDouble** ontvangt een object dat de functionele interface ToDoubleFunction (package java.util.function) implementeert. (Deze interface's applyAsDouble methode geeft een double waarde terug.)

6.4. Extra's

6.4.1. flatMap

```
1 public class VbFlatmap {
2     public static void main(String[] args) {
3
4         List<List<String>> namesNested =
5                 Arrays.asList(Arrays.asList("Jeff", "Bezos"), Arrays.asList("Bill",
  "Gates"),Arrays.asList("Mark", "Zuckerberg"));
6         System.out.println(namesNested);
7
8         List<String> namesFlatStream =
9                 namesNested.stream()
10                            .flatMap(Collection::stream) ①
11                            .collect(Collectors.toList());
12        System.out.println(namesFlatStream);
13    }
14 }
```

① flatMap kan een complexe datastructuur Stream<List<String>> **afvlakken** voor verdere eenvoudige operaties.

```
[[Jeff, Bezos], [Bill, Gates], [Mark, Zuckerberg]]
[Jeff, Bezos, Bill, Gates, Mark, Zuckerberg]
```

6.4.2. orElse

```
1 public class VbOrElse {
2     public static void main(String[] args)
3     {
4         List<String> names = Arrays.asList("Brad" , "Kate" , "Kimmy", "Jack" ,
  "Joey");
5
6         String eersteNaamMet3Letters =
7                 names.stream()
8                      .filter(name -> name.length()==3)
9                      .findFirst()
10                     .get(); ①
11        System.out.println(eersteNaamMet3Letters);
12
13        String eersteNaamMet3Letters2 =
14                names.stream()
15                     .filter(name -> name.length()==3)
16                     .findFirst()
17                     .orElse("Geen naam met 3 letters"); ②
18
19        System.out.println(eersteNaamMet3Letters2);
20    }
21 }
```

① Indien geen resultaat aanwezig, dan wordt door **get** een NoSuchElementException gegooid

② Indien geen resultaat aanwezig, dan wordt nu de tekst "Geen naam met 3 letters" teruggegeven. Indien wel een resultaat, dan wordt het resultaat teruggegeven.
De mogelijkheid bestaat ook om bij geen resultaat een zelfgekozen exception te werpen, bv.: **orElseThrow(NoSuchElementException::new)**;

Overzicht streams:

Intermediate `Stream` operations

`filter`	Results in a stream containing only the elements that satisfy a condition.
`distinct`	Results in a stream containing only the unique elements.
`limit`	Results in a stream with the specified number of elements from the beginning of the original stream.
`map`	Results in a stream in which each element of the original stream is mapped to a new value (possibly of a different type)—e.g., mapping numeric values to the squares of the numeric values. The new stream has the same number of elements as the original stream.
`sorted`	Results in a stream in which the elements are in sorted order. The new stream has the same number of elements as the original stream.

Terminal `Stream` operations

`forEach`	Performs processing on every element in a stream (e.g., display each element).
Reduction operations—*Take all values in the stream and return a single value*	
`average`	Calculates the *average* of the elements in a numeric stream.
`count`	Returns the *number of elements* in the stream.
`max`	Locates the *largest* value in a numeric stream.
`min`	Locates the *smallest* value in a numeric stream.
`reduce`	Reduces the elements of a collection to a *single value* using an associative accumulation function (e.g., a lambda that adds two elements).
Mutable reduction operations—*Create a container (such as a collection or* `StringBuilder`*)*	
`collect`	Creates a *new collection* of elements containing the results of the stream's prior operations.
`toArray`	Creates an *array* containing the results of the stream's prior operations.

Terminal `Stream` operations

Search operations

`findFirst`	Finds the *first* stream element based on the prior intermediate operations; immediately terminates processing of the stream pipeline once such an element is found.
`findAny`	Finds *any* stream element based on the prior intermediate operations; immediately terminates processing of the stream pipeline once such an element is found.
`anyMatch`	Determines whether *any* stream elements match a specified condition; immediately terminates processing of the stream pipeline if an element matches.
`allMatch`	Determines whether *all* of the elements in the stream match a specified condition.