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.
- Begrijpen hoe een stream pipeline opgebouwd wordt, vertrekkend van een databron, gebruik makend van intermediate en terminal operations.
- Uitvoeren van operaties op primitieve streams en op object streams.

2. Functioneel programmeren, een inleiding

Functioneel programmeren

Functioneel programmeren is een programmeerparadigma waarbij software wordt ontwikkeld met behulp van **zuivere functies**. Men vermijdt het gebruik van *side-effects*, *mutable data* en *shared state*.

Functioneel programmeren is **declaratief** in plaats van imperatief.

Bij imperatief programmeren ligt de focus vooral op **HOE?**. Je gaat nadenken over de typische stappen die je moet uitvoeren om een resultaat te bekomen en je gaat deze stappen in detail beschrijven. Daartegenover staat declaratief programmeren. Hierbij ligt de focus vooral op **WAT?**. Je gaat minder bezig zijn met de stappen die je moet ondernemen om je doel te bereiken. Een mooi voorbeeld van een declaratieve taal is **SQL**. Als je alle klanten wil die in Gent wonen kan je je ongeveer volgende query voorstellen:

```
select *
from Klant
where Woonplaats = 'Gent'
```

Je specifieert hier dus duidelijk **wat** je wil, zonder dat je je op enige manier moet bezig houden met de stappen die moeten ondernomen worden om de juiste klanten te vinden. Dit laatste laat je over aan het DBMS die dit heel efficiënt en performant doet voor jou.

Ook in deze cursus maakte je reeds kennis met deze programmeerstijl. Om een verzameling te sorteren heb je bijvoorbeeld gebruik gemaakt van de *sort*-methode uit de klasse Collections. Hoewel achter het sorteren van een verzameling een complexe logica schuilgaat, die meerdere iteraties over de verzameling vergt, hoefde je dit niet zelf uit te werken. De *sort*-methode verwacht enkel dat je een implementatie aanlevert van de **functionele** (let op de naam!) **Comparator** interface. **Lambda expressies** lieten toe dat je dit heel beschrijvend, in-line, kon doen. De iteraties die nodig zijn om te sorteren gebeuren impliciet, wij hoeven zelf niet expliciet een lus gebruiken om de verzameling te sorteren:

```
        List<String> namenLijst = Arrays.asList(new String[] { "Ena", "Cho", "Aiko",
"Rai" });
        Collections.sort(namenLijst, (n1, n2) -> n1.compareTo(n2));
        for (String naam : namenLijst) {
            System.out.println(naam);
        }
        // Aiko Cho Ena Rai
```

Merk op dat deze manier van sorteren wel degelijk een side-effect heeft: de oorspronkelijk verzameling, in bovenstaand voorbeeld *namenLijst*, is na het uitvoeren van de sort-methode gewijzigd. Begrijp dat je niet steeds het zuivere functioneel programmeren zoals we hierboven hebben omschreven zal terugvinden in Java. Maar we gaan we in dit hoofdstuk zien dat je met **Java streams** heel dicht in de buurt komt van het zuivere functionele programmeren. Straks maak je kennis met een manier om een verzameling te sorteren zonder side-effects…

- **externe, expliciete iteraties**
  - *in je programma zelf lussen uitschrijven*
  - *sequentiële uitvoering*
  - *gebruik van variabelen (tellers, accumulators)*
- **interne, impliciete iteraties**
  - *de lussen worden achter de schermen uitgevoerd door het framework*
  - *geoptimaliseerd voor parallelle uitvoering*
  - *je declareert geen variabelen voor de iteratie*

3. Functionele interfaces en lamba expressies

In dit hoofdstuk maken we constant gebruik van functionele interfaces en lambda expressies. Een beknopte herhaling van deze concepten vind je hier.

3.1. Lambda expressies

Lambda expressies zijn **anonieme methodes**. Ze worden typisch gebruikt om een implementatie te voorzien voor de abstracte methode uit een **functionele interface**. Via lambda expressies kan je deze implementatie op een heel beknopte manier schrijven. Je hoeft bijvoorbeeld zelf niet expliciet een instantie van een klasse, die de functionele interface implementeert, te maken.

Een lambda expressie bestaat uit een parameterlijst, gevolgd door een pijltoken, gevolgd door een body.

Algemene vorm lambda expressie

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

Volgende lambda expressie heeft twee parameters van het type int en retourneert hun som:

```
(int i, int j) -> { return i + j; }
```

Indien de body enkel bestaat uit het retourneren van een waarde dan mogen de accolades en het return statement weggelaten worden:

```
(int i, int j) -> i + j
```

De types voor de parameters zijn optioneel. Lambda expressies ondersteunen **target typing** wat betekent dat het type van de parameters zal afgeleid worden uit de context waarin de lambda expressie gebruikt wordt.

```
(i, j) -> i + j
```

Indien de parameterlijst slechts 1 parameter bevat dan mogen de haakjes rond de parameter weggelaten worden. Volgende lambda expressie retourneert een boolean die aangeeft of een geheel getal al dan niet even is:

```
i -> i % 2 == 0
```

Indien de lambda expressie een lege parameterlijst heeft dan moet je de ronde haakjes gebruiken aan de linkerkant van het pijltoken.

```
() -> LocalDateTime.now()
```

3.2. Functionele interfaces

Functionele interfaces, ook gekend als Single Abstract Method (SAM) interfaces, bevatten exact één abstracte methode. In het voorbeeld hierboven maakten we gebruik van de **Comparator** interface om de namenlijst te sorteren.

voorbeeld funcionele interface Comparator

```
@FunctionalInterface
public interface Comparator<T>

// de ene abstracte methode:
int compare(T o1, T o2)
```

De package `java.util.function` bevat heel wat **algemene functionele interfaces** die in dit hoofdstuk veel gebruikt zullen worden. Een greep uit de vele functionele interfaces die je in deze package vindt:

Interface	Abstracte methode	Doel	Voorbeeld
`Predicate<T>`	`boolean test(T t)`	controleren of t aan een bepaalde voorwaarde voldoet	`getal → getal > 10`
`Consumer<T>`	`void accept(T t)`	uitvoeren van een bewerking op t (!side effect!)	`string → System.out.println(string)`
`Function<T, R>`	`R apply(T t)`	uitvoeren van een bewerking op t met een resultaat van type R	`string → string.length()`
`Supplier<T>`	`T get()`	ophalen van een resultaat	`() → LocalDateTime.now()`
`UnaryOperator<T>`	`T apply(T t)`	uitvoeren van een bewerking op t met een resultaat van type T	`getal → getal * 2`
`BinaryOperator<T>`	`T apply(T t1, T t2)`	uitvoeren van een bewerking op t1 en t2 met een resultaat van type T	`(getal1, getal2) ) → getal1 + getal2`

4. Streams

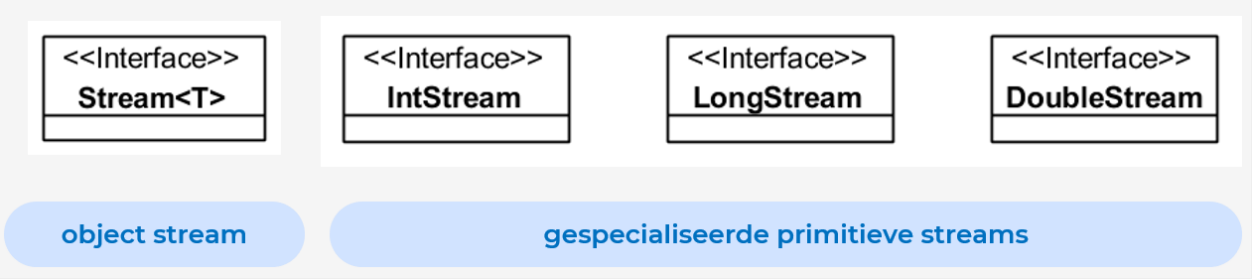
4.1. Basis concepten

Wat?

Een **Stream** is een sequentie van objecten waarop geaggregeerde en parallelle operaties kunnen toegepast worden.

De operaties die je kan uitvoeren op een stream zitten vervat in de interface `Stream<T>`. Naast deze algemene `Stream<T>` voor streams van objecten, bestaan er ook interfaces voor streams die primitieve waarden *(int, long of double)* bevatten.

Stream interfaces

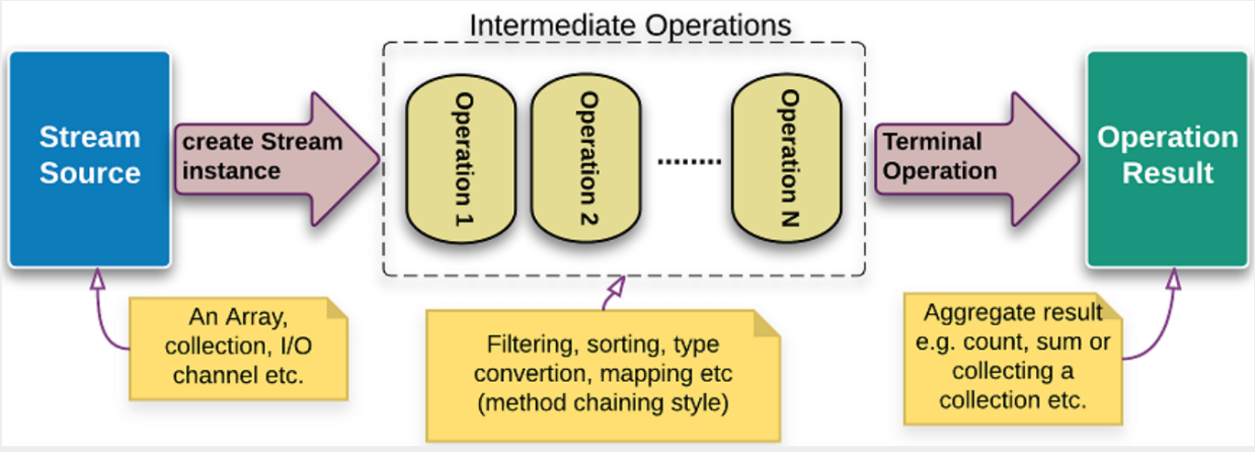


Werken met een stream volgt het **pipeline** principe.

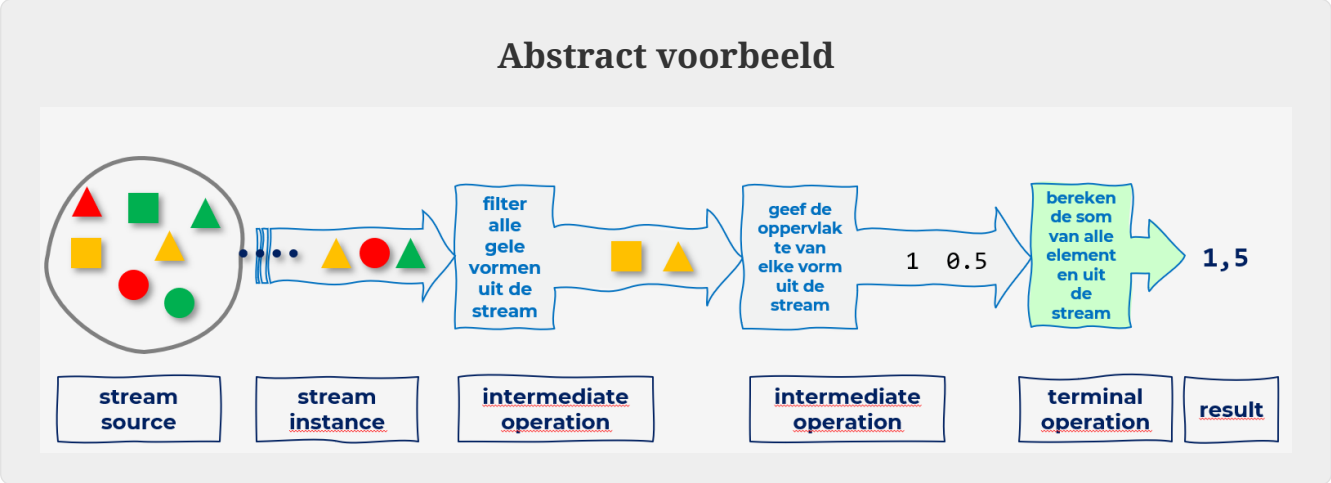
1. creëer een stream vertrekkend van een `databron`
   - databron: een verzameling elementen of een generator die elementen genereert
2. voer 1 of meerdere `intermediate operations` uit
   - intermediate operation: een bewerking die wordt uitgevoerd op de elementen van de stream, als resultaat verkrijg je opnieuw een stream die de bewerkte elementen van de stream bevat
   - chaining: daar het resultaat van een intermediate operation opnieuw een stream is ga je typische verschillende intermediate operations aan elkaar rijgen
3. beëindig de stream met een `terminal operation`
   - terminal operation: een bewerking die wordt uitgevoerd op de elementen van een stream en een resultaat oplevert die niet langer een stream is
   - dit resultaat kan een waarde van eender welk type zijn

een stream kan je niet gebruiken om data bij te houden, als de stream geëvalueerd is via een terminal operation kan je deze niet hergebruiken!

De stream pipeline



Hieronder zie je, zonder concrete Java code te gebruiken, hoe dit principe in zijn werk gaat om voor een verzameling van vormen *de oppervlakte van alle gele vormen* te berekenen.



4.2. Lazy & Eager evaluatie

intermediate operations op een stream volgen het principe van **lazy evaluation**

Dit betekent dat intermediate operations niet zullen worden uitgevoerd wanneer ze gedefinieerd worden. De uitvoering ervan wordt uitgesteld tot de terminal operation wordt aangeroepen.

In onderstaand voorbeeld vindt er bijvoorbeeld geen evaluatie plaats. We vormen een databron om tot een stream en maken daarna gebruik van de intermediate operation `filter`. Deze intermediate operation zal enkel de elementen die voldoen aan de gegeven voorwaarde laten passeren. Maar let op: het omvormen van de databron naar een stream en de filter worden **niet uitgevoerd** daar er geen terminal operation wordt gebruikt.

```
    public static void start() {
        List<String> dataSource = new ArrayList<>(Arrays.asList("yellow", "green",
"red", "orange"));
        Stream<String> stream = dataSource.stream().filter(s -> s.length() > 5);
        dataSource.add("violet");
        // no terminal operation yet...
    }
```

een **terminal operation** op een stream volgt het principe van **eager evaluation**

Dit betekent dat de databron zal worden omgevormd tot een stream, alle intermediate operations zullen worden uitgevoerd, en uiteindelijk ook de terminal operation wordt uitgevoerd om zo tot een resultaat te komen.

In onderstaand voorbeeld beëindigen we de nu de stream met de terminal operation `count`. `count` is een terminal operation die als resultaat een getal retourneert (geen stream!). Zoals je uit de naam kan afleiden zullen we via `count` kunnen te weten komen hoeveel elementen een stream bevat. In onderstaand voorbeeld wordt de databron omgevormd tot een stream wanneer we het resultaat

van `count` toekennen aan de variabele `result`. De elementen van de stream worden gefilterd en uiteindelijk wordt het resultaat berekend. Het element 'violet' behoort op dat moment wel degelijk tot de databron. Dit illustreert het *lazy evaluation* principe: 'violet' behoorde nog niet tot de databron op moment dat de stream werd gedefinieerd.

```
    public static void start() {
        List<String> dataSource = new ArrayList<>(Arrays.asList("yellow", "green",
"red", "orange"));
        Stream<String> stream = dataSource.stream().filter(s -> s.length() > 5);
        dataSource.add("violet");
        long result = stream.count(); // result <- 3
        System.out.printf("Counted %d colors with a long name...%n", result);
    }
```

```
// Uitvoer
Counted 3 colors with a long name...
```

4.3. Streams houden geen data bij

eens een stream werd geëvalueerd via een **terminal operation** is de stream niet meer bruikbaar

Er wordt een IllegalStateException geworpen indien je een stream een tweede maal probeert te gebruiken...

```
    public static void start() {
        List<String> dataSource = new ArrayList<>(Arrays.asList("yellow", "green",
"red", "orange"));
        Stream<String> stream = dataSource.stream().filter(s -> s.length() > 5);
        long result = stream.count();
        long result2 = stream.count(); //IllegalStateException!
    }
```

```
Exception in thread "main" java.lang.IllegalStateException: stream has already been
operated upon or closed
```

5. Van een databron naar een stream...

5.1. Creatie van een IntStream, LongStream & DoubleStream

Maak gebruik van de static method `of` uit de interface `IntStream` om een IntStream te maken

vertrekkend van een integer-array, of een opsomming van gehele getallen. Je kan ook een IntStream maken die alle gehele getallen in een bepaald interval bevat.

```
public static void start() {
    int[] values = { 3, 4, 6, 1 };
    IntStream stream1 = IntStream.of(values); ①
    IntStream stream2 = IntStream.of(-2, 3, 1, 6); ②
    IntStream stream3 = IntStream.range(4, 8); ③
    IntStream stream4 = IntStream.rangeClosed(4, 8); ④
}
```

① stream1 zal de elementen 3, 4, 6 en 1 produceren

② stream2 zal de elementen -2, 3, 1 en 6 produceren

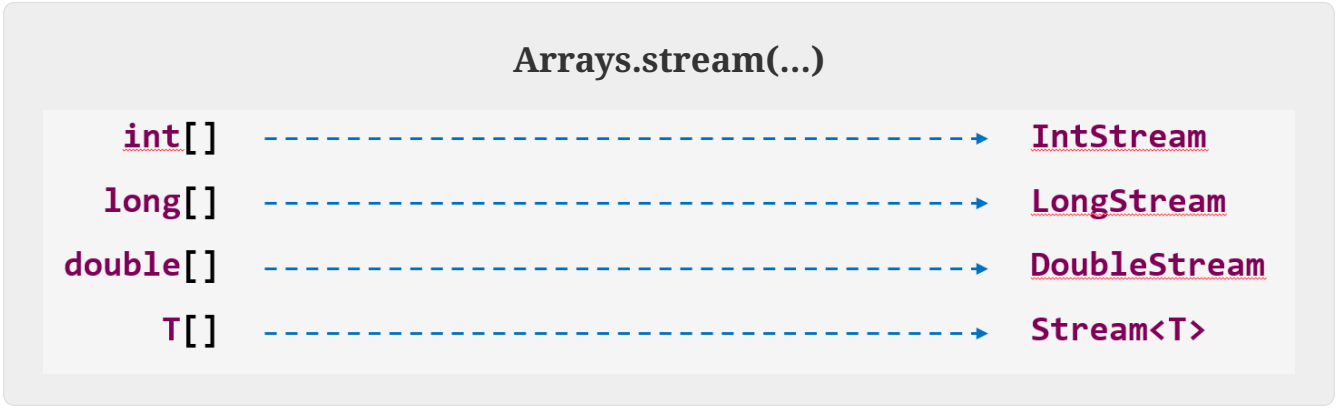
③ stream3 zal de elementen 4, 5, 6 en 7 produceren

④ stream4 zal de elementen 4, 5, 6, 7 en 8 produceren

maak gebruik van gelijkaardige methodes om een **LongStream** of een **DoubleStream** te maken, maar let wel: **DoubleStream** kent **geen** `range(closed)`

5.2. De Arrays.stream methode

Indien de databron voor een stream de elementen uit een array zijn kan je gebruik maken van de statisch methode `stream` uit de klasse `Arrays`. Via deze methode bouw je een stream die alle, of een deel van, de elementen uit de array bevat. Het type stream die je verkrijgt verschilt naargelang het type van de elementen die in de array zitten.




```
public static void start() {
    String[] colors = { "Red", "Orange", "Yellow", "Green", "Blue" };
    Stream<String> colorStream = Arrays.stream(colors); ①
    Stream<String> someColors = Arrays.stream(colors, 1, 3); ②
}
```

① colorStream zal de elementen "Red", "Orange", "Yellow", "Green" en "Blue" produceren

② someColors zal de elementen "Orange" en "Yellow" produceren

5.3. De Collections<E>.stream methode

Ook de interface Collection bevat een statische methode die je toelaat een stream te creëren vertrekkend van de inhoud van een Collection. Analoog aan de stream methode uit Arrays zal het type van de elementen in de Collection bepalend zijn voor het type van de stream.

```
    public static void start() {
       List<Employee> employees = new ArrayList<>();
       boolean succes = Collections.addAll(employees,
           new Employee("Jason", "Red", 5000, "IT"),
           new Employee("Ashley", "Green", 7600, "IT"),
           new Employee("Matthew", "Indigo", 3587.5, "Sales"));

        Stream<Employee> employeeStream = employees.stream(); ①
    }
```

① employeeStream zal de drie Employee-objecten uit de collection employees produceren

6. Enkele veelgebruikte intermediate operations

6.1. filter

Maak gebruik van `filter` indien je van een stream enkel geïnteresseerd bent in de elementen die **voldoen aan een bepaalde voorwaarde**.

De voorwaarde waaraan de elementen moeten voldoen om in de resulterende stream te komen zit vervat in een **predicaat** die je als **argument aan de filter methode** doorgeeft. Dit predicaat is een implementatie van de functionele interface `Predicate` uit de package `java.util.functions`. Je maakte reeds kennis met deze functionele interface in paragraaf 4. De te implementeren methode is een boolse methode, en in de meeste gevallen gaan we deze aanleveren via een lambda expressie.

de filter-methode uit Stream<T>

```
    Stream<T> filter(Predicate<? super T> predicate)
    // Returns a stream consisting of the elements of this stream
    // that match the given predicate.
    // This is an intermediate operation.
```

De functionele interface **Predicate**:

Interface	Abstracte methode	Doel	Voorbeeld
`Predicate<T>`	`boolean test(T t)`	controleren of t aan een bepaalde voorwaarde voldoet	`getal → getal > 10`

Een paar voorbeeldjes:

*Straks gaan we in detail inzoomen op terminal operations maar om de **lazy intermediate `filter` operation** toch reeds in actie te zien maken we hier weerom gebruik van de **terminal operation `count`** die telt hoeveel elementen er in de stream zitten...*

```
    public static void start() {
        int[] values = { 13, 3, 26, 1, -99 };
        long nrOfValuesBiggerThan10 = IntStream.of(values).filter(v -> v > 10).
count();
        System.out.printf("Number of values bigger than 10: %d%n",
nrOfValuesBiggerThan10);
        long nrOfEvenValues = IntStream.of(values).filter(v -> v % 2 == 0).count();
        System.out.printf("Number of even values: %d%n", nrOfEvenValues);

        String[] colors = { "Red", "Brown", "Yellow", "Green", "Blue" };
        long nrOfColorsStartingWithB = Arrays.stream(colors).filter(c -> c.startsWith
("B")).count();
        System.out.printf("Number of colors starting with 'B': %d%n",
nrOfColorsStartingWithB);

        List<Employee> employees = new ArrayList<>();
        boolean succes = Collections.addAll(employees, new Employee("Jason", "Red",
5000, "IT"),
                new Employee("Ashley", "Green", 7600, "IT"), new Employee("Matthew",
"Indigo", 3587.5, "Sales"));
        long nrOfEmployeesEarningAbove1500 = employees.stream().filter(e -> e
.getSalary() > 1500).count();
        System.out.printf("Number of employees earning more than 1500: %d%n",
nrOfEmployeesEarningAbove1500);
    }
```

```
// Uitvoer
Number of values bigger than 10: 2
Number of even values: 1
Number of colors starting with 'B': 2
Number of employees earning more than 1500: 3
```

6.2. map

Maak gebruik van een `map` indien je **elk element uit een stream wil transformeren**

De transformatie die elk element ondergaat is het resultaat van een methode die de **functionele interface** `Function<T, R>` implementeert. Dit betekent dan ook dat de inkomende stream van type `Stream<T>` wijzigt naar een uitgaande stream van type `Stream<R>`.

Een paar voorbeeldjes:

In deze voorbeeldjes leer je ook kennis maken met de terminal operation **`forEach`**, *via deze operation kan je een actie uitvoeren op elk element van de stream. In onderstaande voorbeeldjes gebruiken we dit om elk element van een stream naar de console weg te schrijven…*

```
    public static void start() {

        // Elk getal verdubbelen
        System.out.printf("-- De getallen verdubbeld --%n");
        int[] values = { 13, 3, 26, 1, -99 };
        IntStream.of(values).map(v -> v * 2).forEach(v -> System.out.printf("%d ",
v));
        System.out.println();

        // Elke string transformeren naar zijn lengte ①
        System.out.printf("%n-- De lengte van de strings --%n");
        String[] colors = { "Red", "Brown", "Yellow", "Green", "Blue" };
        Arrays.stream(colors).map(c -> c.length()).forEach(v -> System.out.printf("%d
", v));
        System.out.println();

        // Elk Employee object transformeren naar zijn naam ②
        System.out.printf("%n-- De namen van de employees --%n");
        List<Employee> employees = new ArrayList<>();
        boolean succes = Collections.addAll(employees, new Employee("Jason", "Red",
5000, "IT"),
                new Employee("Ashley", "Green", 7600, "IT"), new Employee("Matthew",
"Indigo", 3587.5, "Sales"));
        employees.stream().map(e -> e.getName()).forEach(n -> System.out.printf("%s
", n)); ③
    }
```

① Stream<String> wordt IntStream

② Stream<Employee> wordt Stream<String>

③ merk op dat `getName` de voornaam en achternaam van een employee concateneert en retourneert

```
// Uitvoer
-- De getallen verdubbeld --
26 6 52 2 -198

-- De lengte van de strings --
3 5 6 5 4

-- De namen van de employees --
Jason Red   Ashley Green   Matthew Indigo
```

6.3. distinct

Maak gebruik van `distinct` indien je **enkel unieke elementen uit een stream** wil.

Deze methode heeft geen parameters en maakt gebruik van de `equals` methode om te beslissen of twee elementen uit de stream al dan niet gelijk zijn.

Een paar voorbeeldjes:

```
    public static void start() {

        System.out.printf("-- Unieke getallen --%n");
        int[] values = { 13, 3, 13, 13, -13 };
        IntStream.of(values).distinct().forEach(v -> System.out.printf("%d ", v));
        System.out.println();

        System.out.printf("%n-- Unieke kleuren --%n");
        String[] colors = { "Yellow", "Brown", "yellow", "Yellow", "Brown" };
        Arrays.stream(colors).distinct().forEach(c -> System.out.printf("%s ", c));
        System.out.println();

        System.out.printf("%n-- Unieke employees --%n");
        Employee jason = new Employee("Jason", "Red", 5000, "IT");
        Employee jason2 = jason;
        List<Employee> employees = new ArrayList<>();
        Collections.addAll(employees, jason, new Employee("Jason", "Red", 5000, "IT"),
                new Employee("Ashley", "Green", 7600, "IT"), new Employee("Matthew",
"Indigo", 3587.5, "Sales"),
                jason2);
        employees.stream().distinct().forEach(e -> System.out.printf("%s%n", e)); ①
②
    }
```

① het is belangrijk dat je weet hoe de equals methode in Employee is geïmplementeerd om te begrijpen wat hier gebeurt, twee employees zijn gelijk indien hun name en firstname gelijk zijn, zie hieronder...

② in Employee werd de `toString` methode overschreven, zie hieronder...

```
    public class Employee {
        private String firstName;
        private String lastName;
        private double salary;
        private String department;

    // parts of code are omitted

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Employee other = (Employee) obj;
        if (firstName == null) {
            if (other.firstName != null)
                return false;
        } else if (!firstName.equals(other.firstName))
            return false;
        if (lastName == null) {
            if (other.lastName != null)
                return false;
        } else if (!lastName.equals(other.lastName))
            return false;
        return true;
    }

    @Override
    public String toString() {
        return String.format("%-8s %-8s %8.2f   %s", getFirstName(), getLastName(),
getSalary(), getDepartment());
    }
    }
```

```
// Uitvoer
-- Unieke getallen --
13 3 -13

-- Unieke kleuren --
Yellow Brown yellow

-- Unieke employees --
Jason    Red       5000,00   IT
Ashley   Green     7600,00   IT
Matthew  Indigo    3587,50   Sales
```

6.4. sorted

Maak gebruik van `sorted` indien je **de elementen uit een verzameling wenst te sorteren, zonder de oorspronkelijke verzameling te wijzigen (i.e. zonder side effect)**.

Deze methode kan je gebruiken zonder parameters. In dit geval worden elementen gesorteerd op **natuurlijke wijze**. Dit veronderstelt dat de elementen van een type zijn die de interface `Comparable` implementeert.

Je kan de methode ook aanroepen met een `Comparator` en op deze manier een **totale ordening** van de elementen verkrijgen. De interfaces `Comparable` en `Comparator` kwamen in het hoofdstuk "Polymorfisme en Interfaces" uitgebreid aan bod.

Een paar voorbeeldjes ter verduidelijking. Bekijk alvast eerst enkele implementatie details van `Employee`:

```
public class Employee implements Comparable<Employee> {
    private String firstName;
    private String lastName;
    private double salary;
    private String department;

    // parts of code are omitted

    @Override
    public int compareTo(Employee e) {
        int lastNameCompare = lastName.compareTo(e.lastName);
        return (lastNameCompare == 0) ? firstName.compareTo(e.firstName) :
lastNameCompare;
    }
}
```

```
    public static void start() {

        System.out.printf("-- De gesorteerde getallen --%n");
        int[] values = { 13, 3, 50, 1, -13 };
        IntStream.of(values).sorted().forEach(v -> System.out.printf("%d ", v));
        System.out.println();

        System.out.printf("%n-- De gesorteerde kleuren --%n");
        String[] colors = { "Yellow", "Brown", "Green", "Blue", "Red" };
        Arrays.stream(colors).sorted().forEach(c -> System.out.printf("%s ", c));
        System.out.println();

        System.out.printf("%n-- De gesorteerde employees: sorteren op natuurlijke
wijze --%n");
        List<Employee> employees = new ArrayList<>();
        Collections.addAll(employees, new Employee("Jason", "Red", 5000, "IT"),
                new Employee("Ashley", "Green", 7600, "IT"), new Employee("Matthew",
"Indigo", 3587.5, "Sales"));
        employees.stream().sorted().forEach(e -> System.out.printf("%s%n", e));

        System.out.printf("%n-- De gesorteerde employees: totale ordening, dalend op
salary --%n");
        employees.stream().sorted(Comparator.comparing(Employee::getSalary).
reversed())
                .forEach(e -> System.out.printf("%s%n", e));

    }
```

```
// Uitvoer
-- De gesorteerde getallen --
-13 1 3 13 50

-- De gesorteerde kleuren --
Blue Brown Green Red Yellow

-- De gesorteerde employees: sorteren op natuurlijke wijze --
Ashley   Green     7600,00   IT
Matthew  Indigo    3587,50   Sales
Jason    Red       5000,00   IT

-- De gesorteerde employees: totale ordening, dalend op salary --
Ashley   Green     7600,00   IT
Jason    Red       5000,00   IT
Matthew  Indigo    3587,50   Sales
```

7. Enkele terminal operations

Je maakte reeds kennis met de terminal operations `count` en `forEach`. Terminal operations beëindigen een stream.

- sommige **terminal operations** retourneren een **resultaat**
  - als voorbeeld gebruikten we `count`
- sommige **terminal operations** hebben een **side-effect**
  - als voorbeeld gebruikten we `forEach`

We bespreken nu verderop nog enkele interessante terminal operations.

7.1. sum

Maak gebruik van `sum` om **de som van de elementen van een stream getallen te berekenen**. Deze terminal operation is gedefinieerd op `IntStream`, `DoubleStream` en `LongStream`.

Voor streams van het type `Stream<T>` kan je steeds één van de verschillende intermediate `map` operations gebruiken om een bepaalde sommatie uit te voeren. Enkele voorbeeldjes verduidelijken dit principe.

```
    public static void start() {
        int[] values = { 1, 2, 3, 4, 5 };
        long sum = IntStream.of(values).sum();
        System.out.printf("Sum equals %d%n", sum);
        long sumOfSquares = IntStream.of(values).map(v -> (int) Math.pow(v, 2)).sum();
        System.out.printf("Sum of squares equals %d%n", sumOfSquares);

        String[] colors = { "Red", "Yellow", "Green" };
        long numberOfLetters = Arrays.stream(colors).mapToInt(c -> c.length()).sum();
        System.out.printf("Number of letters in our colors: %d%n", numberOfLetters);

        List<Employee> employees = new ArrayList<>();
        Collections.addAll(employees, new Employee("Jason", "Red", 5000, "IT"),
                new Employee("Ashley", "Green", 7600, "IT"), new Employee("Matthew",
"Indigo", 3587.5, "Sales"));
        double totalSalary = employees.stream().mapToDouble(e -> e.getSalary()).sum();
        System.out.printf("Total salaries equals %.2f%n", totalSalary);
    }
```

```
// Uitvoer
Sum equals 15
Sum of squares equals 55
Number of letters in our colors: 14
Total salaries equals 16187,50
```

7.2. Optional types

Er zijn heel wat terminal stream operations die een `Optional` retourneren. In dit deeltje staan we even stil bij de Java Optional klassen, alvorens we de draad terug oppikken met terminal operations die hiervan gebruik maken.

7.2.1. OptionalInt

Een `OptionalInt` is een container object dat **al dan niet een int waarde bevat**. Via de boolse methode `isPresent()` kunnen we achterhalen of het container object een waarde bevat.

Enkele methodes uit de klasse OptionalInt:

```
int getAsInt()
// If a value is present, returns the value,
// otherwise throws NoSuchElementException.

int orElse(int other)
// If a value is present, returns the value,
// otherwise returns other.

boolean isPresent()
// If a value is present, returns true, otherwise false.
```

Analoog aan deze klasse bestaan er ook de klassen `OptionalDouble` en `OptionalLong`.

7.2.2. Optional<T>

Een `Optional<T>` is een container object dat **al dan niet een niet-null waarde bevat**. Via de boolse methode `isPresent()` kunnen we achterhalen of het container object een niet-null waarde bevat.

Enkele methodes uit de klasse Optional<T>:

```
T get()
// If a value is present, returns the value,
// otherwise throws NoSuchElementException.

T orElse(T other)
// If a value is present, returns the value,
// otherwise returns other.

boolean isPresent()
// If a value is present, returns true, otherwise false.

boolean isEmpty()
// If a value is not present, returns true, otherwise false.

<U> Optional<U> map(Function<? super T,? extends U> mapper)
// If a value is present, returns an Optional describing
// the result of applying the given mapping function to the value,
// otherwise returns an empty Optional.
```

7.3. min, max & average

Zoals je uit de namen van deze methodes kan afleiden gebruik je deze methodes om het **minimum**, **maximum** of **gemiddelde** van een stream van getallen te berekenen.

Deze methodes retourneren een **optional**. De optional zal leeg zijn indien de methodes gebruikt worden op een stream die geen elementen bevat.

Je kan min en max ook gebruiken op een **Stream<T>**. De methodes retourneren dan het minimum of maximum element uit de stream volgens een opgegeven **comparator**.

```
    public static void start() {
        int[] values = { 1, -2, 3, 4, -5 };
        System.out.printf("Mininum equals %d%n", IntStream.of(values).min().
getAsInt());
        System.out.printf("Maximum equals %d%n", IntStream.of(values).max().
getAsInt());
        System.out.printf("Average equals %.2f%n", IntStream.of(values).average
().getAsDouble());

        String[] colors = { "Red", "Yellow", "Green" };
        System.out.printf("Longest color is: %s%n",
                Arrays.stream(colors).max(Comparator.comparing(String::length)).
get());

        List<Employee> employees = new ArrayList<>();
        Collections.addAll(employees, new Employee("Jason", "Red", 5000, "IT"),
                new Employee("Ashley", "Green", 7600, "IT"), new Employee("Matthew",
"Indigo", 3587.5, "Sales"));
        Optional<Employee> lowestPaidEmployee = employees.stream().min(Comparator
.comparing(Employee::getSalary));
        System.out.printf("The lowest paid employee is %s%n", lowestPaidEmployee.
get());
        System.out.printf("The name of the lowest paid employee is %s%n",
                lowestPaidEmployee.map(Employee::getName).get());
        employees = new ArrayList<>();
        System.out.printf("The name of the lowest paid employee is %s%n", employees
.stream()
                .min(Comparator.comparing(Employee::getSalary)).map(Employee::
getName).orElse("!No employee found!"));
    }
```

```
// Uitvoer
Maximum equals 4
Average equals 0,20
Longest color is: Yellow
The lowest paid employee is Matthew  Indigo    3587,50   Sales
The name of the lowest paid employee is Matthew Indigo
The name of the lowest paid employee is !No employee found!
```

7.4. allMatch, anyMatch

allMatch(predicate) - gebruik deze boolse methode om na te gaan of **alle elementen** in een stream voldoen aan het **predicaat**

anyMatch(predicate) - gebruik deze boolse methode om na te gaan of er **minstens 1 element** in een stream voldoet aan het **predicaat**

```
    public static void start() {
        int[] values = { 1, -2, 3, 4, -5 };
        System.out.printf("There is %s element bigger than 100%n",
                IntStream.of(values).anyMatch(e -> e > 100) ? "an" : "no");
        System.out.printf("%s elements are smaller than 100%n",
                IntStream.of(values).allMatch(e -> e < 100) ? "All" : "Not all");

        String[] colors = { "Red", "Yellow", "Green" };
        System.out.printf("Our colors do %scontain Yellow%n",
                Arrays.stream(colors).anyMatch(c -> c.equals("Yellow")) ? "" : "not
");

        List<Employee> employees = new ArrayList<>();
        Collections.addAll(employees, new Employee("Jason", "Red", 5000, "IT"),
                new Employee("Ashley", "Green", 7600, "IT"), new Employee("Matthew",
"Indigo", 3587.5, "Sales"));
        System.out.printf("There is %s employee by the name of Green%n",
                employees.stream().anyMatch(e -> e.getLastName().equals("Green")) ?
"an" : "no");
        System.out.printf("There is %s employee by the name of Java%n",
                employees.stream().anyMatch(e -> e.getLastName().equals("Java")) ?
"an" : "no");
    }
```

```
// Uitvoer
There is no element bigger than 100
All elements are smaller than 100
Our colors do contain Yellow
There is an employee by the name of Green
There is no employee by the name of Java
```

7.5. reduce

Gebruik een reduce om alle elementen van een stream te reduceren tot **1 samenvattend element**.

Je moet aan deze methode een implementatie van de functionele interface **BinaryOperator** doorgeven.

Interface	Abstracte methode	Doel	Voorbeeld
`BinaryOperator<T>`	`T apply(T t1, T t2)`	uitvoeren van een bewerking op t1 en t2 met een resultaat van type T	`(getal1, getal2) ) → getal1 + getal2`

Het resultaat van een reduce is een `Optional`. Voor een goede werking van de reduce moet je er ook voor zorgen dat de binary operator associatief is. De optelling is een voorbeeld van een associatieve

binaire operator (let op de haakjes):

```
(20 + 5) + 2 == 20 + (5 + 2)
```

De deling is een voorbeeld van een niet associatieve binaire operator:

```
(20 / 5) / 2 != 20 / (5 / 2)
```

Je kan ook een **initiële waarde** voorzien voor een reducer. Deze waarde moet een **identity** waarde zijn voor de gebruikte binary operator. Dit betekent dat als je de operator gebruikt met de identity waarde en een andere waarde, de andere waarde ongewijzigd blijft. 0 is een voorbeeld van een identity waarde voor de plus, de lege string is een identity waarde voor string concatenatie:

```
0 + 10 == 10

"" + "abc" == "abc"
```

Als je een initiële waarde voorziet voor de reducer is het resultaat niet langer een optional; ook al zijn er geen elementen in de stream dan verkrijg je alsnog de initiële waarde als resultaat van de reducer.

```
    public static void start() {
        int[] values = { 1, -2, 3, 4, -5 };
        System.out.printf("Sum of all values via reduce method: %d%n",
                IntStream.of(values).reduce((i1, i2) -> i1 + i2).getAsInt());
        System.out.printf("Sum of all even values via reduce method: %d%n",
                IntStream.of(values).filter(i -> i % 2 == 0).reduce((i1, i2) -> i1 +
i2).getAsInt());

        String[] colors = { "Red", "Green", "Blue" };
        System.out.printf("First letters of our colors: %s%n",
                Arrays.stream(colors).reduce("", (c1, c2) -> c1 + c2.substring(0,
1)));

        List<Employee> employees = new ArrayList<>();
        Collections.addAll(employees, new Employee("Jason", "Red", 5000, "IT"),
                new Employee("Ashley", "Green", 7600, "IT"), new Employee("Matthew",
"Indigo", 3587.5, "Sales"));
        System.out.printf("Initials of all our employees: %s",
                employees.stream().map(e -> e.getFirstName().substring(0, 1) + e
.getLastName().substring(0, 1))
                        .reduce("", (s1, s2) -> s1 + s2 + " "));
    }
```

```
// Uitvoer
Sum of all values via reduce method: 1
Sum of all even values via reduce method: 2
First letters of our colors: RGB
Initials of all our employees: JR AG MI
```

7.6. toArray

Via de **toArray** methode kan je de elementen van een stream verzamelen in array. Dit is een belangrijke terminal operator!

Dikwijls zal je een array gebruiken als databron voor een stream. Zoals je al in dit hoofdstuk hebt gemerkt kan je door deze array om te zetten naar een stream gebruik maken van beknopte declaratieve code om complexe en krachtige bewerkingen op je data los te laten. Met de `toArray` methode ben je nu in staat om elementen uit een stream terug te verzamelen in een array.

Als je de `toArray` methode gebruikt om elementen te verzamelen uit een `Stream<T>` ga je typisch een generator doorgeven. Deze wordt gebruikt om de resulterende array te alloceren. Meestal volstaat het hier een aanroep te doen naar de `new` methode voor een array van het gewenste type. Om elementen te verzamelen uit een primitieve stream (IntStream, DoubleStream, LongStream) hoef je dit niet te doen.

```
    public static void start() {
        int[] values = { 1, -2, 3, 4, -5 };
        double[] squaredValuesOfEvenNumbers = IntStream.of(values).filter(i -> i % 2
== 0)
                .mapToDouble(i -> Math.pow(i, 2)).toArray();
        System.out.printf("Resulting array with squares of even numbers%n");
        for (double d : squaredValuesOfEvenNumbers) {
            System.out.printf("%.2f   ", d);
        }
        System.out.printf("%n%n");

        String[] colors = { "Red", "Green", "Blue", "Violet", "Black", "White",
"Orange" };
        String[] sortedColors = Arrays.stream(colors).sorted().toArray(String[]::new);
        System.out.printf("Resulting array with sorted colors%n");
        for (String c : sortedColors) {
            System.out.printf("%s   ", c);
        }
        System.out.printf("%n%n");

        List<Employee> employees = new ArrayList<>();
        Collections.addAll(employees, new Employee("Jason", "Red", 5000, "IT"),
                new Employee("Ashley", "Green", 7600, "IT"), new Employee("Matthew",
"Indigo", 3587.5, "Sales"));
        System.out.printf("Array with employees of IT department: %n");
        Employee[] itEmployees = employees.stream().filter(e -> e.getDepartment
().equals("IT"))
                .toArray(Employee[]::new);
        for (Employee e : itEmployees) {
            System.out.printf("%s%n", e);
        }
        System.out.println();
    }
```

```
// Uitvoer
Resulting array with squares of even numbers
4,00   16,00

Resulting array with sorted colors
Black   Blue   Green   Orange   Red   Violet   White

Array with employees of IT department:
Jason    Red       5000,00   IT
Ashley   Green     7600,00   IT
```

7.7. toList

Via de **toList** methode, gedefinieerd op `Stream<T>` kan je de elementen van een stream verzamelen in een `List`.

Deze methode is niet gedefinieerd voor primitieve streams. De resulterende `List` is niet te wijzigen. je kan aan die lijst bijvoorbeeld geen elementen toevoegen of verwijderen. Verderop zullen we zien dat we via Collectors ook elementen kunnen verzamelen in een List die wel te wijzigen is.

```
    public static void start() {
        String[] colors = { "Red", "Green", "Blue", "Violet", "Black", "White",
"Orange" };
        List<String> sortedColors = Arrays.stream(colors).sorted().toList();
        System.out.printf("Resulting list with sorted colors%n");
        for (String c : sortedColors) {
            System.out.printf("%s   ", c);
        }
        System.out.printf("%n%n");

        List<Employee> employees = new ArrayList<>();
        Collections.addAll(employees, new Employee("Jason", "Red", 5000, "IT"),
                new Employee("Ashley", "Green", 7600, "IT"), new Employee("Matthew",
"Indigo", 3587.5, "Sales"));
        System.out.printf("Array with employees of IT department: %n");
        List<Employee> itEmployees = employees.stream().filter(e -> e.getDepartment
().equals("IT")).toList();
        for (Employee e : itEmployees) {
            System.out.printf("%s%n", e);
        }
        System.out.println();
    }
```

```
// Uitvoer
Resulting list with sorted colors
Black   Blue   Green   Orange   Red   Violet   White

Array with employees of IT department:
Jason    Red       5000,00   IT
Ashley   Green     7600,00   IT
```

7.8. collect

De `collect` methode laat toe om op een heel veelzijdige manier elementen van een stream te verzamelen in een container.

Als je de collect methode gebruikt moet je een Collector doorgeven. Deze collector hoef je zelf niet

te schrijven want de klasse `Collectors` bevat heel wat interessante collectors. Deze zijn er gedefinieerd als statische methodes. We geven hieronder enkele voorbeelden...

```
    public static void start() {
        String[] colors = { "Red", "Green", "Green", "Violet" };
        System.out.printf("Our colors in 1 string: %s%n%n",
                Arrays.stream(colors).sorted().collect(Collectors.joining(" - ")));
        Set<String> uniqueColors = Arrays.stream(colors).collect(Collectors.toSet());
        System.out.printf("Colors collected in a set, quite unique! %n");
        for (String c : uniqueColors) {
            System.out.printf("%s%n", c);
        }
        System.out.printf("%n");

        List<Employee> employees = new ArrayList<>();
        Collections.addAll(employees, new Employee("Jason", "Red", 5000, "IT"),
                new Employee("Ashley", "Green", 7600, "IT"), new Employee("Matthew",
"Indigo", 3587.5, "Sales"));
        System.out.printf("Modifiable list with IT employees: %n");
        List<Employee> itEmployees = employees.stream().filter(e -> e.getDepartment
().equals("IT"))
                .collect(Collectors.toList());
        for (Employee e : itEmployees) {
            System.out.printf("%s%n", e);
        }
        System.out.printf("%nAll employees in 1 string: %n");
        System.out.printf("%s", employees.stream().map(Employee::toString).collect
(Collectors.joining("\n")));
    }
```

```
// Uitvoer
Our colors in 1 string: Green - Green - Red - Violet

Colors collected in a set, quite unique!
Red
Violet
Green

Modifiable list with IT employees:
Jason    Red       5000,00   IT
Ashley   Green     7600,00   IT

All employees in 1 string:
Jason    Red       5000,00   IT
Ashley   Green     7600,00   IT
Matthew  Indigo    3587,50   Sales
```

8. Tot slot…

In dit hoofdstuk maakte je kennis met de kracht van functioneel programmeren. Hoewel we hebben ingezoomd op talrijke intermediate en terminal operations bestaan er nog veel meer operations die in dit hoofdstuk niet aan bod zijn gekomen. Indien je gegevens in een verzameling moet bewerken ga je nu nog zelden algoritmes concreet moeten uitschrijven met behulp van lussen en variabelen. Neem altijd eens een kijkje in de Java documentatie alvorens je veel werk en energie stopt in een algoritme die je meer dan waarschijnlijk declaratief kan uitschrijven als -een combinatie van- stream operations. Op een gepaste wijze stream operations kunnen combineren om tot een gewenst resultaat te komen vergt natuurlijk veel oefening. Waar wacht je op? Veel stream plezier!