



samen sterk voor werk

Java

() -> { LAMBDA }



Deze cursus is eigendom van VDAB Competentiecentra ©

Peoplesoftcode:

Wettelijk depot:

versie: 17/3/2015

INHOUD

1	Inleiding	3
1.1	Doelstelling.....	3
1.2	Vereiste voorkennis.....	3
1.3	Nodige software	3
2	Basis	4
2.1	Algemeen.....	4
2.2	Syntax van een lambda	4
2.2.1	Algemeen	4
2.2.2	Verkortingen op de algemene syntax.....	4
2.3	Een lambda is een implementatie van een functional interface	4
2.4	@FunctionalInterface.....	5
3	Voorbeelden van lambda gebruik in de standaard libraries.....	6
3.1	Voorbeeld 1: Comparator	6
3.2	Voorbeeld 2: Runnable.....	6
3.3	Voorbeeld 3: ActionListener.....	8
4	Optional	10
4.1	Voorbeeld van het probleem met null.....	10
4.2	NullPointerException aanpakken met Optional.....	10
5	Stream en forEach	12
5.1	forEach.....	12
6	Databronnen van een Stream.....	13
6.1	List	13
6.2	Set.....	13
6.3	Map.....	13
6.4	String	14
6.5	Tekstbestand	14
6.6	Directory.....	14
6.7	Enkele losse waarden	15
6.8	Een oneindige reeks waarden	15
6.9	Een getallenreeks	16
7	Filteren	17
8	Sorteren	18

9	Unieke waarden	20
10	Transformeren	21
11	Controleren of één of alle waarden in een Stream voldoen aan een voorwaarde.....	23
11.1	allMatch	23
11.2	anyMatch	23
12	Stream waarden verzamelen.....	24
12.1	Array.....	24
12.2	List.....	25
12.3	Set	25
12.4	Map	25
12.5	String.....	26
13	Statistieken (count, sum, min, max, average).....	27
13.1	count	27
13.2	statistieken op int waarden	27
13.2.1	sum.....	27
13.2.2	min, max en average	27
13.2.3	Van een gewone Stream naar een IntStream.....	28
13.3	statistieken op long waarden	28
13.4	statistieken op double waarden	28
13.5	min en max op waarden die geen int, long of double zijn	28
13.6	sum op waarden die geen int, long of double zijn	29
14	Waarden in de Stream reduceren tot één waarde	30
15	Meerdere streams combineren tot één stream	32
16	Parallel stream	33
17	Method references.....	35
18	Herhalings oefeningen	36

1 Inleiding

1.1 Doelstelling

Je leert lambda's gebruiken in je Java code.

Java bevat lambda's sinds Java 8.

Je code wordt met lambda's kort en leesbaar.

1.2 Vereiste voorkennis

- Java Programming Fundamentals

1.3 Nodige software

- een JDK (Java Developer Kit) met versie 8 of hoger
- Een Java IDE (NetBeans, Eclipse, ...)

2 Basis

2.1 Algemeen

Lambda's bestaan in veel programmeertalen.

Een lambda is in elke taal een anonieme functie: een functie zonder naam.

Een lambda kan wel de onderdelen hebben die ook functies mét een naam hebben

- parameter(s)
- code
- optioneel een returnwaarde

Het symbool λ stelt een lambda voor.

2.2 Syntax van een lambda

2.2.1 Algemeen

```
(parameter1, parameter2, ...) -> { code van de functie }
```

Voorbeeld: een lambda met twee parameters die deze parameters afbeeldt:

```
(int getal, char teken) -> { System.out.println(getal); System.out.println(teken); }
```

2.2.2 Verkortingen op de algemene syntax

Parametertypes zijn in een lambda optioneel. Het vorige voorbeeld kan dus korter:

```
(getal, teken) -> { System.out.println(getal); System.out.println(teken); }
```

Bij een lambda met één parameter mag je de ronde haakjes rond die parameter weglaten:

```
getal -> { System.out.println(getal); }
```

Bij een lambda zonder parameters vermeld je een open en een sluit rond haakje:

```
() -> { System.out.println("Hallo"); }
```

Als een lambda één opdracht bevat,

mag je de accolades rond die opdracht en de ; na die opdracht weglaten:

```
getal -> System.out.println(getal)
```

Als een lambda maar één opdracht bevat én dit is een **return** opdracht,

mag je ook het sleutelwoord **return** weglaten.

Voorbeeld: een lambda die het kwadraat van een parameter teruggeeft:

```
getal -> getal * getal
```

Dit laatste voorbeeld toont duidelijk het streven naar compactheid in een lambda.

2.3 Een lambda is een implementatie van een functional interface

Een functional interface is een interface die één abstracte method bevat.

Een lambda is een implementatie van een functional interface.

Je maakt in je IDE (NetBeans, Eclipse, ...) een project voor een console applicatie.

Je voegt aan dit project de functional interface EvenGetallen toe

```
public interface EvenGetallen {  
    boolean isEven(int getal);  
}
```

Je voegt een class Main toe, waarin je met een lambda de functional interface implementeert. Het eerste voorbeeld is eenvoudig en dient enkel om lambda's te leren kennen. Je ziet verder in de cursus veel praktische voorbeelden van lambda's.

```
class Main {
    public static void main(String[] args) {
        EvenGetallen evenGetallen =
            getal -> getal % 2 == 0;
        System.out.println(evenGetallen.isEven(7));
    }
}
```

- (1) Het type van de variabele evenGetallen is de functional interface EvenGetallen.
- (2) Je kan de variabele daarom invullen met een lambda (als implementatie van de interface). De parameter signatuur van de lambda moet overeenstemmen met de parameter signatuur van de abstract method in de functional interface.
 - a. Die method heeft één parameter, dus de lambda heeft ook één parameter. De parameter heeft in de method int als type. De compiler controleert of je in de lambda enkel handelingen op de parameter uitvoert die je op een int kan uitvoeren.
 - b. De method geeft een boolean terug. De compiler controleert of je lambda ook een boolean teruggeeft.
- (3) Je voert de method isEven, beschreven in de interface uit, en je geeft als parameterwaarde 7 mee. Java voert hierbij de lambda uit, geeft de waarde 7 mee als parameter van de lambda en geeft de returnwaarde van de lambda terug als returnwaarde van de method isEven.

2.4 @FunctionalInterface

Zodra een interface meer dan één abstract method bevat, is dit geen functional interface meer. Je kan die interface dus niet meer implementeren met een lambda.

Je probeert dit uit: je voegt volgende abstract method toe aan de interface EvenGetallen:

```
boolean isOnEven(int getal);
```

Zodra je de source opslaat, zie je een compilerfout in de lambda in de class Main.

Je verwijdert de abstract method isOnEven terug uit de interface EvenGetallen.

Je tik @FunctionalInterface voor een functional interface om te verhinderen dat die interface niet voldoet aan de vereiste van een functional interface (één abstract method bevatten).

Zodra de interface niet voldoet aan die vereiste, produceert de compiler een fout in de interface zelf.

```
@FunctionalInterface
public interface EvenGetallen {
    boolean isEven(int getal);
}
```

Als je nu probeert een abstract method aan de interface toe te voegen, krijg je een compilerfout.

3 Voorbeelden van lambda gebruik in de standaard libraries

Sommige Java libraries bevatten al lang functional interfaces. Je vindt hier drie voorbeelden.

3.1 Voorbeeld 1: Comparator

De interface Comparator is een functional interface: hij bevat één abstracte method:

```
int compare(T object1, T object2);
```

Je moet dus geen class moet schrijven om deze interface te implementeren, maar je kan een korte, krachtige lambda schrijven als implementatie van de interface.

De method compare moet een int teruggeven. Deze moet

- negatief zijn als `object1 < object2`
- 0 zijn als `object1 == object2`
- positief zijn als `object1 > object2`

Je probeert dit in de class Main: je sorteert een String array van Z naar A

```
import java.util.Arrays;
```

```
class Main {
    public static void main(String[] args) {
        String[] groenten = { "tomaat", "sla", "ui", "prei" };
        Arrays.sort(groenten,
            (groente1, groente2) -> -groente1.compareTo(groente2));
        System.out.println(Arrays.toString(groenten));
    }
}
```

❶

- (1) De sort method heeft twee parameters. De eerste is de te sorteren array. De tweede is een implementatie van de functional interface Comparator. Je geeft hier een lambda mee. Deze heeft twee parameters, zoals de compare method in de interface. De lambda geeft een int terug, zoals de compare method in Comparator. Deze int is de negatie (-) van de int die compareTo method van de String class teruggeeft. De compareTo method vergelijkt twee Strings alfabetisch (A naar Z). Je bekomt met de negatie een vergelijking in omgekeerde volgorde (Z naar A). Het sorteer algoritme in de sort method gebruikt deze vergelijking tijdens het sorteren.

Je kan dit uitproberen.

Gewijzigd voorbeeld: je sorteert op het aantal tekens in de groente.

Je wijzigt de code in de lambda

```
groente1.length() - groente2.length()
```

Je kan dit uitproberen.

3.2 Voorbeeld 2: Runnable

De interface Runnable stelt code voor die je in een thread wil uitvoeren.

Runnable is een functional interface, want hij bevat één abstracte method:

```
void run();
```

Je moet dus geen class schrijven om deze interface te implementeren, maar je kan een korte, krachtige lambda schrijven als implementatie van de interface.

De voorbeeldapplicatie moet twee taken uitvoeren

1. De lege regels verwijderen uit `countries.txt`
2. Dubbele waarden verwijderen uit `languages.txt`

Je maakt de applicatie performant door elke taak in een aparte gelijktijdige thread uit te voeren.

Je kopieert `countries.txt` en `languages.txt` in een folder `data` in de root van de harddisk.

Je wijzigt de class Main:

```
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.Writer;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.LinkedHashSet;
import java.util.Set;
class Main {
    private final static Path DATA_PATH = Paths.get("/data");
    private final static Path COUNTRIES_PATH =
        DATA_PATH.resolve("countries.txt");
    private final static Path COUNTRIES_BACKUP_PATH =
        DATA_PATH.resolve("countries.bak");
    private final static Path LANGUAGES_PATH =
        DATA_PATH.resolve("languages.txt");
    private final static Path LANGUAGES_BACKUP_PATH =
        DATA_PATH.resolve("languages.bak");
    private static void legeRegelsVerwijderen() {
        try {
            Files.deleteIfExists(COUNTRIES_BACKUP_PATH);
            Files.move(COUNTRIES_PATH, COUNTRIES_BACKUP_PATH);
            try (BufferedReader reader = Files.newBufferedReader(COUNTRIES_BACKUP_PATH);
                Writer bufferedWriter = Files.newBufferedWriter(COUNTRIES_PATH);
                PrintWriter writer = new PrintWriter(bufferedWriter)) {
                for (String regel; (regel = reader.readLine()) != null;) {
                    if (!regel.isEmpty()) {
                        writer.printf("%s\n", regel);
                    }
                }
            }
        } catch (Exception ex) { ex.printStackTrace(); }
    }
    private static void dubbelsVerwijderen() {
        Set<String> uniekeTalen = new LinkedHashSet<>();
        try {
            Files.deleteIfExists(LANGUAGES_BACKUP_PATH);
            Files.move(LANGUAGES_PATH, LANGUAGES_BACKUP_PATH);
            try (BufferedReader reader=Files.newBufferedReader(LANGUAGES_BACKUP_PATH)) {
                for (String regel; (regel = reader.readLine()) != null;) {
                    uniekeTalen.add(regel);
                }
            }
            try (Writer bufferedWriter = Files.newBufferedWriter(LANGAGES_PATH);
                PrintWriter writer = new PrintWriter(bufferedWriter)) {
                for (String taal : uniekeTalen) {
                    writer.printf("%s\n", taal);
                }
            }
        } catch (Exception ex) { ex.printStackTrace(); }
    }
    public static void main(String[] args) {
        new Thread(() -> legeRegelsVerwijderen()).start();
        new Thread(() -> dubbelsVerwijderen()).start();
    }
}
```

- (1) Deze constante bevat het pad naar de folder data in de root van de harddisk.

- (2) Deze constante combineert het pad naar de folder data in de root van de harddisk met het pad naar het bestand countries.txt in die folder tot één pad : /data/countries.txt
- (3) Je hernoemt countries.txt naar countries.bak
- (4) Je leest regel per regel uit countries.bak
- (5) Als een regel niet leeg is
- (6) schrijf je hem weg naar countries.txt
- (7) Je leest regel per regel uit languages.bak
- (8) Je voegt die toe aan de Set, die de dubbele waarden elimineert.
- (9) Je schrijf de unieke waarden uit de Set één per één weg naar languages.txt
- (10) Het type van de parameter van de constructor van de class Thread is de functional interface Runnable. Je kan dus een lambda meegeven als implementatie van Runnable.
De lambda heeft geen parameters, zoals de method run in Runnable.
De lambda bevat maar één opdracht, daarom moeten er geen accolades rond die opdracht en moet er ook geen ; na die opdracht.

Je kan dit uitproberen.

3.3 Voorbeeld 3: ActionListener

ActionListener stelt code voor die Java uitvoert als een gebruiker een button aanklikt in een Swing applicatie. ActionListener is een functional interface, want hij bevat één abstracte method:

```
void actionPerformed(ActionEvent e);
```

Je moet dus geen class schrijven om deze interface te implementeren, maar je kan een korte, krachtige lambda schrijven als implementatie van de interface.

Het voorbeeld bevat een textbox en twee buttons. De eerste button converteert de tekst in de textbox naar hoofdletters. De tweede button converteert de tekst in de textbox naar kleine letters.

Je wijzigt de class Main:

```
import java.awt.BorderLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;
class Venster extends JFrame {                                ❶
    private static final long serialVersionUID = 1L;
    Venster() {
        super("Conversie");                                    ❷
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);          ❸
        JTextField textField = new JTextField();                ❹
        JButton buttonHoofdLetters = new JButton("hoofdletters"); ❺
        JButton buttonKleineLetters = new JButton("kleine letters");
        buttonHoofdLetters.addActionListener(
            e -> textField.setText(textField.getText().toUpperCase()); ❻
        buttonKleineLetters.addActionListener(
            e -> textField.setText(textField.getText().toLowerCase());
        add(textField, BorderLayout.NORTH);                      ❼
        add(buttonHoofdLetters, BorderLayout.WEST);              ❽
        add(buttonKleineLetters, BorderLayout.EAST);             ❾
        pack();
    }
}
class Main {
    public static void main(String[] args) {
        new Venster().setVisible(true);                          ❿
    }
}
```

- (1) Een class die een GUI venster voorstelt erft van JFrame.

- (2) Je roept de constructor van de base class (JFrame) op.
Java toont de tekst die je meegeeft in de titelbalk van je venster.
- (3) Je geeft aan dat wanneer de gebruiker het venster sluit, het volledige programma stopt.
- (4) Je maakt een invoervak waarin de gebruiker tekst kan tikken.
- (5) Je maakt een knop met als tekst hoofdletters.
- (6) Het type van de parameter van de method `addActionListener` is de functional interface `ActionListener`. Je kan dus een lambda meegeven als implementatie van `ActionListener`.
De lambda heeft één parameter, zoals de method `actionPerformed` in `ActionListener`.
De lambda bevat maar één opdracht, daarom moeten er geen accolades rond die opdracht en moet er ook geen `;` na die opdracht.
- (7) Je plaatst het invoervak boven in het venster.
- (8) Je plaatst de knop links in het venster.
- (9) Je plaatst de knop rechts in het venster.
- (10) Je toont het venster.

Je kan de applicatie uitproberen.



Hoger lager: zie takenbundel

4 Optional

Java bevat, zoals veel programmeertalen, het sleutelwoord **null** om het 'niets' voor te stellen.

Jammer genoeg leidt het gebruik van **null** regelmatig tot een `NullPointerException`. Deze treedt op als je op een reference variabele die **null** bevat toch een handeling doet, zoals een method uitvoeren, ...

Sinds Java 8 bestaat de class `Optional`, die als doel heeft `NullPointerExceptions` te vermijden. Je leert hier deze class kennen, en hoe ze samenwerkt met lambda's.

4.1 Voorbeeld van het probleem met null

Je maakt een method waarmee je collega's het eerste cijfer in een tekst zoeken.

Een eerste idee over de method signatuur:

```
int eersteCijfer(String string)
```

Maar als de parameter `string` geen enkel cijfer bevat, kan je geen `int` teruggeven.

Een tweede idee over de method signatuur:

```
Integer eersteCijfer(String string)
```

De returnwaarde bevat

- het eerste cijfer in de parameter `string`, als die parameter één of meerdere cijfers bevat
- **null** als de parameter `string` geen enkel cijfer bevat

Het probleem is dat je collega met het returntype (`Integer`) geen *duidelijke* hint krijgt dat de returnwaarde **null** kan zijn.

Je wijzigt de class `Main` om dit te zien:

```
class Main {
    private static Integer eersteCijfer(String string) {
        for (int index = 0; index != string.length(); index++) {
            char teken = string.charAt(index);
            if (Character.isDigit(teken)) {
                return Character.getNumericValue(teken);
            }
        }
        return null;
    }
    public static void main(String[] args) {
        System.out.println(eersteCijfer("all4you") * 10);
        System.out.println(eersteCijfer("wrong") * 10);
    }
}
```

(1) Je krijgt hier een `NullPointerException`, want je kan `null` niet vermenigvuldigen met `10`.

4.2 NullPointerException aanpakken met Optional

Sinds Java 8, beschrijf je het returntype van een method, die ook 'niets' kan teruggeven, als een `Optional`. `Optional` geeft aan je collega's *expliciet* aan dat de returnwaarde 'iets' maar ook 'niets' kan bevatten.

Je wijzigt de class Main:

```
import java.util.Optional;
class Main {
    private static Optional<Integer> eersteCijfer(String string) {           ❶
        for (int index = 0; index != string.length(); index++) {
            char teken = string.charAt(index);
            if (Character.isDigit(teken)) {                                  ❷
                return Optional.of(Character.getNumericValue(teken));
            }
        }
        return Optional.empty();                                           ❸
    }
    public static void main(String[] args) {
        Optional<Integer> optioneelCijfer = eersteCijfer("all4you");         ❹
        if (optioneelCijfer.isPresent()) {                                   ❺
            System.out.println(optioneelCijfer.get() * 10);                 ❻
        }
        optioneelCijfer = eersteCijfer("wrong");
        if (optioneelCijfer.isPresent()) {
            System.out.println(optioneelCijfer.get() * 10);
        }
    }
}
```

- (1) Optional is een generic type.
Je geeft tussen < en > het type mee van de waarde die de Optional optioneel kan bevatten.
- (2) Er bestaat geen public constructor om een Optional te maken.
Je kan wel een Optional maken met de static method of.
Je geeft aan deze method de waarde mee die je in de Optional wil plaatsen.
- (3) Je maakt een Optional die 'niets' bevat met de static method empty.
- (4) De programmeur die de method oproept ziet aan het returntype (Optional) expliciet dat de returnwaarde ook 'niets' kan bevatten en beseft (hopelijk) dat hij daar rekening moet mee houden.
- (5) De method isPresent() geeft true terug als de Optional een waarde bevat en geeft false terug als de Optional 'niets' bevat.
- (6) De method get() geeft je de waarde in de Optional.

Je kan dit uitproberen.

Dankzij lambda's kan je de code in de main method korter schrijven:

```
Optional<Integer> optioneelCijfer = eersteCijfer("all4you");
optioneelCijfer.ifPresent(cijfer -> System.out.println(cijfer * 10));       ❶
optioneelCijfer = eersteCijfer("wrong");
optioneelCijfer.ifPresent(cijfer -> System.out.println(cijfer * 10));
```

- (1) De method ifPresent aanvaardt een lambda.
De method ifPresent voert die lambda enkel uit als de Optional een waarde bevat.
De lambda krijgt die waarde binnen als parameter.

Uiteindelijk kan de code nog korter:

```
eersteCijfer("all4you").ifPresent(cijfer -> System.out.println(cijfer * 10));
eersteCijfer("wrong").ifPresent(cijfer -> System.out.println(cijfer * 10));
```



Landcodes: zie takenbundel

5 Stream en forEach

De interface `Stream` is nieuw sinds Java 8. Deze interface stelt een datastroom voor.

Deze datastroom heeft een bron nodig waaruit de datastroom ontstaat.

Deze bron kan een array zijn, een `List`, een `Set`, de regels uit een tekstbestand, ...

Je ziet van al deze bronnen voorbeelden in deze cursus.

Je kan op deze datastroom interessante operaties uitvoeren met methods uit de interface `Stream`:

- sorteren
- filteren
- statistische informatie (minimum, maximum, som, ...)
- ...

De methods uit de interface `Stream` aanvaarden een lambda als parameter.

Dit leidt ook hier tot korte, leesbare code.

5.1 `forEach`

Je voert met de `forEach` method een handeling uit op elk waarde uit de `Stream`.

Je wijzigt de class `Main`:

```
import java.util.Arrays;
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        String[] groenten = { "tomaat", "sla", "ui", "prei" };
        Stream<String> stream = Arrays.stream(groenten);
        stream.forEach(groente -> System.out.println(groente));
    }
}
```

❶
❷
❸

- (1) Deze array zal de bron worden van een `Stream`.
- (2) Je geeft aan de `stream` method van de class `Arrays` een array mee.
De `stream` method maakt een `Stream` gebaseerd op de waarden in die array.
`Stream` is een generic interface met tussen `<` en `>` het type van de waarden in de `Stream`.
- (3) Je geeft aan de method `forEach` een lambda mee.
De method `forEach` roept die lambda op per waarde in de `Stream`
en geeft die waarde mee als parameter van de lambda.
Jij beslist in de lambda code wat je met deze waarde doet.

Je kan dit uitproberen.

De laatste twee opdrachten worden regelmatig tot één opdracht gecombineerd:

```
Arrays.stream(groenten)
    .forEach(groente -> System.out.println(groente));
```

6 Databronnen van een Stream

6.1 List

Een List kan de databron zijn van een Stream. Je wijzigt de class Main:

```
import java.util.Arrays;
import java.util.List;
class Main {
    public static void main(String[] args) {
        List<String> groenten = Arrays.asList("tomaat", "sla", "ui", "prei");
        groenten.stream()
            .forEach(groente -> System.out.println(groente));
    }
}
```

(1) De method stream van List maakt een Stream gebaseerd op de waarden in de List.

6.2 Set

Een Set kan de databron zijn van een Stream. Je wijzigt de class Main:

```
import java.util.LinkedHashSet;
import java.util.Set;
class Main {
    public static void main(String[] args) {
        Set<Integer> heiligeGetallen = new LinkedHashSet<>();
        heiligeGetallen.add(1);
        heiligeGetallen.add(3);
        heiligeGetallen.add(4);
        heiligeGetallen.add(7);
        heiligeGetallen.stream()
            .forEach(getal -> System.out.println(getal));
    }
}
```

(1) De method stream van Set maakt een Stream gebaseerd op de waarden in de Set.

6.3 Map

De keySet, de values of de entrySet van een Map kunnen de databron zijn van een Stream. Je gebruikt als voorbeeld de entrySet in de class Main.

```
import java.util.LinkedHashMap;
import java.util.Map;
class Main {
    public static void main(String[] args) {
        Map<String, String> talen = new LinkedHashMap<>();
        talen.put("NL", "Nederlands");
        talen.put("FR", "Frans");
        talen.entrySet().stream()
            .forEach(entry -> System.out.println(entry.getKey() + ':' +
                entry.getValue()));
    }
}
```

(1) De method stream van Set maakt een Stream gebaseerd op de waarden in de Set.

6.4 String

Een String kan de databron zijn van een Stream.

De waarden in de Stream zijn de tekens in de String. Je wijzigt de class Main:

```
class Main {
    public static void main(String[] args) {
        "Lambdafun".chars()
            .forEach(unicode->System.out.println((char) unicode));
    }
}
```

- (1) De method chars van String maakt een Stream gebaseerd op de tekens in de String.
- (2) Je krijgt één teken niet als een char, maar als een int met de unicode waarde van het teken. Je converteert dit terug naar een char.

6.5 Tekstbestand

Een tekstbestand kan de databron zijn van een Stream.

De waarden in de Stream zijn Strings met de regels in het tekstbestand. Je wijzigt de class Main:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.stream.Stream;
class Main {
    private final static Path LANGUAGES_PATH = Paths.get("/data/languages.txt");
    public static void main(String[] args) {
        try (Stream<String> stream = Files.lines(LANGUAGES_PATH)) {
            stream.forEach(regel -> System.out.println(regel));
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

- (1) Je geeft aan de lines method van de class Files het pad mee naar een tekstbestand. De method geeft een Stream<String> terug met één waarde per regel uit het tekstbestand.

6.6 Directory

Een directory kan de databron zijn van een Stream.

De waarden in de Stream zijn de paden van de bestanden en subdirectory's in die directory.

Je wijzigt de class Main:

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.stream.Stream;
class Main {
    private final static Path DATA_PATH = Paths.get("/data");
    public static void main(String[] args) {
        try (Stream<Path> stream = Files.list(DATA_PATH)) {
            stream.forEach(entry -> System.out.println(entry.getFileName()));
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```


- (1) Je geeft aan de `list` method van de class `Files` het pad mee naar een directory. De method geeft je een `Stream<Path>` terug met één pad per bestand en subdirectory uit de directory.

6.7 Enkele losse waarden

Enkele losse waarden kunnen de databron zijn van een `Stream`. Je wijzigt de class `Main`:

```
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        Stream.of("Adam", "Eva")
            .forEach(naam -> System.out.println(naam));
    }
}
```

❶

- (1) Je geeft aan de `Stream` method `of()` enkele waarden mee. (hier als voorbeeld de waarden Adam en Eva). De method geeft je een `Stream` terug met de waarden die je meegaf aan `of()`



6.8 Een oneindige reeks waarden

Een oneindige reeks waarden kan de databron zijn van een `Stream`. Je wijzigt de class `Main`:

```
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        Stream.iterate(1, vorigGetal -> vorigGetal + 2)
            .forEach(onevenGetal -> System.out.println(onevenGetal));
    }
}
```

❶

- (1) Je geeft aan de method `iterate` twee parameters mee. De eerste parameter is de eerste waarde uit een reeks waarden. De tweede parameter is een lambda. De parameter van die lambda is de vorige waarde uit de reeks. Jij geeft in de lambda de volgende waarde in de reeks terug (in dit geval de vorige waarde + 2). In ons voorbeeld is de eerste waarde uit de reeks 1. De tweede waarde is de vorige waarde (1) + 2, dus 3. De derde waarde is de vorige waarde (3) + 2, dus 5. ... Je bekomt dus de oneven getallen.

Je kan dit uitproberen. Gezien de reeks oneindig is, stopt het programma niet vanzelf en zal je het moeten afbreken met  rechts onder in NetBeans of  naast het tabblad Console in Eclipse.

Je kan dit omvormen naar een *eindige* reeks met de `Stream` method `limit`.

```
Stream.iterate(1, vorigGetal -> vorigGetal + 2)
    .limit(10)
    .forEach(onevenGetal -> System.out.println(onevenGetal));
```

❶

- (1) Je geeft aan de `Stream` method `limit` het maximaal aantal waarden in de `Stream` mee. De `limit` method geeft een `Stream` terug met dit maximaal aantal waarden. zodat je op die `Stream` een andere `Stream` method (`forEach`) kan toepassen.

6.9 Een getallenreeks

Een getallenreeks kan de databron zijn van een Stream. Je wijzigt de class Main:

```
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        IntStream.rangeClosed(1, 10)
            .forEach(getal->System.out.println(getal));
        IntStream.range(1, 10)
            .forEach(getal->System.out.println(getal));
    }
}
```

❶

❷

- (1) Je geeft aan de method `rangeClosed` twee parameters mee. Je krijgt een reeks gehele getallen vanaf de eerste parameter tot en met de tweede parameter.
- (2) Je geeft aan de method `range` twee parameters mee. Je krijgt een reeks gehele getallen vanaf de eerste parameter tot (exclusief) de tweede parameter.



Sterrenbeelden: zie takenbundel

7 Filteren

Je wil niet altijd *alle* waarden uit de Stream verwerken.

Je behoudt enkel de waarden in de Stream die je *wél* wil verwerken met de Stream method `filter`.

Je wijzigt de class `Main`:

```
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        Stream<String> groenten = Stream.of("sla", "wortel", "kool", "biet");
        Stream<String> stream = groenten.filter(
            groente -> groente.length() == 4);
        stream.forEach(groenteMet4Letters -> System.out.println(groenteMet4Letters));
    }
}
```

- (1) Je geeft aan de method `filter` een lambda mee.
 De method `filter` roept die lambda op per waarde in de Stream
 en geeft die waarde mee als parameter van de lambda.
 Je lambda moet `true` teruggeven als deze waarde in de Stream mag blijven
 (omdat je deze waarde wil verwerken).
 De filter method geeft je een Stream waarin enkel de gefilterde waarden overblijven.

Je kan dit uitproberen.

De opdrachten worden regelmatig tot één opdracht gecombineerd:

```
Stream.of("sla", "wortel", "kool", "biet")
    .filter(groente -> groente.length() == 4)
    .forEach(groenteMet4Letters -> System.out.println(groenteMet4Letters));
```

Sommige programmeurs vrezen dat de performantie van deze code slecht is.

Ze denken dat Java eerst de `filter` method uitvoert en hierin itereert over de oorspronkelijke groenten en een verzameling met `kool` en `biet` opbouwt in het geheugen. Ze denken dat Java daarna itereert over deze verzameling en per groente de lambda bij `forEach` uitvoert.

Deze vrees is ongegrond. Java itereert maar één keer over de oorspronkelijke groenten. Binnen die iteratie controleert Java of een groente voldoet aan de conditie in de lambda in de `filter` method. Als dit het geval is, voert Java de lambda uit in de `forEach` method.

Een grappige voorstelling van de method `filter` (uit de oorspronkelijke verzameling met 4 elementen worden enkel de vegetarische overgehouden):

```
filter([🍔, 🍟, 🍗, 🍿], isVegetarian)
=> [🍟, 🍿]
```



Zoek een sterrenbeeld: zie takenbundel

8 Sorteren

Je sorteert de waarden in de Stream met de Stream method `sorted`.

Je wijzigt de class Main:

```
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        Stream.of("sla", "wortel", "kool", "biet")
            .sorted()
            .forEach(groente -> System.out.println(groente));
    }
}
```

❶

- (1) De method `sorted` sorteert de waarden in de Stream in 'natural' order. Bij byte, short, int of long waarden is dit sorteren van klein naar groot. Bij char waarden is dit alfabetisch sorteren. Bij objecten is dit sorteren op basis van de `compareTo` method van die objecten. De `compareTo` method van String objecten vergelijkt Strings op hun alfabetische volgorde.

Je kan dit uitproberen.

Er bestaat ook een versie van de `sorted` method waarmee je kan sorteren op een andere volgorde dan de 'natural' order. Je geeft aan die versie een lambda mee. Deze lambda implementeert de functional interface `Comparator`. De `sorted` method sorteert dan gebaseerd op deze lambda.

Je wijzigt de class Main: je sorteert op het aantal tekens in de groenten:

```
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        Stream.of("sla", "wortel", "kool", "biet")
            .sorted((groente1, groente2) -> groente1.length() - groente2.length())
            .forEach(groente -> System.out.println(groente));
    }
}
```

Je kan dit uitproberen.

Je kan ook sorteren op *meerdere* criteria. Je sorteert als voorbeeld de groenten op het aantal tekens. Je sorteert groenten met hetzelfde aantal tekens alfabetisch.

```
import java.util.Comparator;
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        Comparator<String> comparator =
            (groente1, groente2) -> groente1.length() - groente2.length();
        comparator = comparator.thenComparing(
            (groente1, groente2) -> groente1.compareTo(groente2));
        Stream.of("sla", "wortel", "kool", "biet")
            .sorted(comparator)
            .forEach(groente -> System.out.println(groente));
    }
}
```

❶

❷

❸

- (1) Bij sorteren op *meerdere* criteria moet je de lambda, die de functional interface `Comparator` implementeert, voorbereidend toekennen aan een variabele.
- (2) Je voert op die variabele de method `thenComparing` uit en je geeft opnieuw een lambda mee die de functional interface `Comparator` implementeert. Waarden in de Stream die volgens de lambda bij (1) gelijk zijn, worden gesorteerd op basis van deze tweede lambda.
- (3) Je geeft de combinatie van beide lambda's mee aan de `sorted` method.

Je kan dit uitproberen.

Je combineert in het volgende voorbeeld filter, sorted en forEach:

```
import java.util.stream.Stream;
public class Main {
    public static void main(String[] args) {
        Stream.of("sla", "wortel", "kool", "biet")
            .filter(groente -> groente.length() == 4)
            .sorted()
            .forEach(groente -> System.out.println(groente));
    }
}
```



Oneven getallen: zie takenbundel

9 Unieke waarden

De Stream method `distinct` geeft je een Stream met de unieke waarden uit de oorspronkelijke Stream waarop je de method `distinct` uitvoerde.

Je wijzigt de class Main:

```
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        Stream.of("sla", "kool", "wortel", "biet", "sla")
            .distinct()
            .forEach(groente -> System.out.println(groente));
    }
}
```

Je kan dit uitproberen.

10 Transformeren

De Stream method `map` geeft je een Stream met evenveel waarden als de oorspronkelijke Stream waarop je de method `map` uitvoerde. De waarden in de nieuw Stream zijn anders dan de waarden in de oorspronkelijke Stream. Jij geeft een lambda mee aan de method `map`, waarmee je beschrijft hoe je een waarde uit de oorspronkelijke Stream transformeert naar een waarde in de nieuwe Stream.

Je transformeert als voorbeeld een Stream met `sla`, `wortel`, `kool` en `biet` naar een Stream met de lengtes van die groenten: 3, 6, 4, 4

Je wijzigt de class `Main`:

```
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        Stream.of("sla", "wortel", "kool", "biet")
            .map(groente -> groente.length())
            .forEach(lengte -> System.out.println(lengte));
    }
}
```

- (1) Je geeft aan de method `map` een lambda mee. De method `map` roept die lambda op per waarde in de Stream en geeft die waarde mee als parameter van de lambda. Jij geeft een getransformeerde waarde terug. De method `map` geeft een Stream terug met alle getransformeerde waarden.

Het volgende voorbeeld toont een gesorteerde lijst van oppervlakten van rechthoeken:

```
import java.util.stream.Stream;
class Rechthoek {
    private final int lengte;
    private final int breedte;
    Rechthoek(int lengte, int breedte) {
        this.lengte = lengte;
        this.breedte = breedte;
    }
    int getOppervlakte() {
        return lengte * breedte;
    }
}
class Main {
    public static void main(String[] args) {
        Stream.of(new Rechthoek(6, 2), new Rechthoek(3, 1), new Rechthoek(5, 4))
            .map(rechthoek -> rechthoek.getOppervlakte())
            .sorted()
            .forEach(oppervlakte -> System.out.println(oppervlakte));
    }
}
```

- (1) Je transformeert iedere rechthoek naar zijn oppervlakte.
- (2) Je sorteert deze oppervlakten.

Een grappige voorstelling van de method `map` (de 4 elementen uit de oorspronkelijke verzameling worden via de transformatie "cook" omgezet naar voedsel)





Artiestennamen: zie takenbundel



Landnamen sorteren: zie takenbundel

11 Controleren of één of alle waarden in een Stream voldoen aan een voorwaarde

11.1 allMatch

De Stream method `allMatch` geeft `true` terug als *alle* waarden in de Stream voldoen aan een voorwaarde. Anders geeft de method `allMatch` `false` terug. Je geeft de voorwaarde mee als de parameter van de method `allMatch`.

Je wijzigt de class `Main`: je controleert of alle groenten bestaan uit 4 tekens:

```
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        System.out.println(
            Stream.of("sla", "wortel", "kool", "biet")
                .allMatch(groente -> groente.length() == 4));
    }
}
```

- (1) Je geeft aan de method `allMatch` een lambda mee. De method `allMatch` voert die lambda uit per waarde in de Stream en geeft die waarde mee als parameter van de lambda. Jij geeft een boolean expressie terug die aangeeft waaraan de waarde moet voldoen. Als die expressie bij *alle* waarden `true` teruggeeft, geeft de method `allMatch` `true` terug.

Je kan dit uitproberen.

11.2 anyMatch

De Stream method `anyMatch` geeft `true` terug als minstens één waarde in de Stream voldoet aan een voorwaarde. Anders geeft de method `anyMatch` `false` terug.

Je geeft de voorwaarde mee als de parameter van de method `anyMatch`.

Je wijzigt de class `Main`: je controleert of minstens één groente bestaat uit 4 tekens:

```
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        System.out.println(
            Stream.of("sla", "wortel", "kool", "biet")
                .anyMatch(groente -> groente.length() == 4));
    }
}
```

- (2) Je geeft aan de method `anyMatch` een lambda mee. De method `anyMatch` voert die lambda uit per waarde in de Stream en geeft die waarde mee als parameter van de lambda. Jij geeft een boolean expressie terug die aangeeft waaraan de waarde moet voldoen. Zodra die expressie bij één waarde `true` teruggeeft, geeft de method `anyMatch` `true` terug.

Je kan dit uitproberen.

12 Stream waarden verzamelen

De laatste handeling die je tot nu op een Stream uitvoerde was de `forEach` method. Je leert hier dat je als laatste handeling de waarden in de Stream kan verzamelen in een array, List, Set, Map of String

12.1 Array

Je kan als laatste handeling op een Stream de waarden in de Stream verzamelen in een array met de Stream method `toArray`.

Je geeft aan die method een lambda mee. Deze lambda krijgt als parameter het aantal waarden in de Stream binnen. Jij geeft in de lambda een array terug met evenveel elementen als dit aantal. De method `toArray` vult die array met de waarden in de Stream en geeft je die array terug.

Je wijzigt de class Main: je toont met Swing de gesorteerde sterrenbeelden uit `sterrenbeelden.txt`

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.stream.Stream;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
class Venster extends JFrame {
    private static final long serialVersionUID = 1L;
    private static final Path STERRENBEELDEN_PATH =
        Paths.get("/data/sterrenbeelden.txt");
    Venster() {
        super("Sterrenbeelden");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JList<String> listSterrenbeelden;
        try (Stream<String> stream = Files.lines(STERRENBEELDEN_PATH)) {
            listSterrenbeelden = new JList<>(
                stream.sorted()
                    .toArray(aantalElementen -> new String[aantalElementen]));
        } catch (IOException ex) {
            JOptionPane.showMessageDialog(this, "Kan sterrenbeelden niet lezen");
            listSterrenbeelden = new JList<>();
        }
        add(new JScrollPane(listSterrenbeelden));
        setExtendedState(JFrame.MAXIMIZED_BOTH);
    }
}
class Main {
    public static void main(String[] args) {
        new Venster().setVisible(true);
    }
}
```

- (1) Je geeft aan de JList constructor een array mee. De JList toont de elementen in die array.
- (2) De parameter van de lambda is het aantal waarden in de Stream.
Jij maakt een array met evenveel elementen.
De method `toArray` vult die array met de waarden in de Stream en geeft je die array.

Je kan dit uitproberen.

12.2 List

Je kan als laatste handeling op een Stream de waarden in de Stream verzamelen in een List met de Stream method collect.

Je wijzigt de class Main:

```
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;
class Main {
    public static List<String> gesorteerdeGroenten() {
        return Stream.of("sla", "wortel", "kool", "biet")
            .sorted()
            .collect(Collectors.toList());
    }
    public static void main(String[] args) {
        System.out.println(gesorteerdeGroenten());
    }
}
```

❶

- (1) Je geeft aan de collect method een object mee dat de interface Collector implementeert. Zo'n object helpt de collect method om de waarden in de Stream ter verzamelen. De class Collectors bevat enkele handige methods die zo'n object teruggeven. De method toList geeft een object terug dat de method collect helpt de waarden in de Stream ter verzamelen in een List. De method collect geeft je die List als returnwaarde.

Je kan dit uitproberen.

12.3 Set

De Collectors method toSet werkt zoals de method toList, maar geeft een object terug dat de method collect helpt de waarden in de Stream te verzamelen in een Set.

De Stream method collect(Collectors.toSet()) geeft dus een Set terug.

12.4 Map

Je leert de waarden uit een Stream groeperen op een gemeenschappelijke waarde en deze groepen op te nemen in een Map. Je doet dit met de Stream method collect.

Je zal de groenten sla, kool, wortel en biet groeperen op de lengte van iedere groente:

```
3 sla
4 kool biet
6 wortel
```

Je wijzigt de class Main:

```
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        Map<Integer, List<String>> groentenPerLengte =
            Stream.of("sla", "kool", "wortel", "biet")
                .collect(Collectors.groupingBy(
                    groente -> groente.length()));
        groentenPerLengte.entrySet().stream()
            .forEach(entry -> {System.out.print(entry.getKey());
                entry.getValue().stream()
                .forEach(groente -> System.out.print(groente));
                System.out.println();});
    }
}
```

❶

❷

❸

❹

- (1) Je groepeerde de waarden uit een Stream met de method `collect`.
Je geeft aan de `collect` method een object mee dat de interface `Collector` implementeert.
Zo'n object helpt de `collect` method om de waarden in de Stream te verzamelen.
De class `Collectors` bevat enkele handige methods die zo'n object teruggeven.
De method `groupingBy` geeft een object terug dat de method `collect` helpt de waarden in de Stream te verzamelen in groepen. De method `collect` geeft je die groepen als een `Map`.
- (2) Je geeft aan de method `groupingBy` een lambda mee.
De method `groupingBy` voert die lambda uit per waarde in de Stream en geeft die waarde mee als parameter van de lambda.
Jij geeft in de lambda een waarde terug waarop je de waarden in de Stream wil groeperen.
Dit is hier de lengte van iedere String met een groente.
De `collect` method geeft je een `Map` terug.
De key van iedere Entry zijn de waarden waarop je groepeerde: de lengtes 3, 4, 6.
De value van iedere Entry is een verzameling met de waarden uit de oorspronkelijke Stream die horen bij de key van de Entry.
De `Map` in ons voorbeeld bevat drie entries:
 - a. key: 3 value: List met één waarde: sla
 - b. key: 4 value: List met twee waarden: kool en biet
 - c. key: 6 value: List met één waarde: wortel
- (3) Je toont per Entry de key van de Entry.
- (4) De value van de Entry bevat een List. Je toont elke waarde in die List.

Je kan dit uitproberen.

12.5 String

Je kan als laatste handeling op een Stream met String waarden deze waarden opnemen in één String met de Stream method `collect`.

Je wijzigt de class `Main`:

```
import java.util.stream.Collectors;
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        System.out.println(Stream.of("sla", "wortel", "kool", "biet")
                                .collect(Collectors.joining(", ")));
    }
}
```

❶

- (1) Je geeft aan de `collect` method een object mee dat de interface `Collector` implementeert.
Zo'n object helpt de `collect` method om de waarden in de Stream te verzamelen.
De class `Collectors` bevat enkele handige methods die zo'n object teruggeven.
De method `joining` geeft een object terug dat de method `collect` helpt de waarden in de Stream te verzamelen in één String. De method `collect` geeft je die String als returnwaarde. Je geeft aan de method `joining` een String mee.
De method `collect` gebruikt deze String als scheidingsteken tussen de waarden bij het opbouwen van de samengestelde String.

Je kan dit uitproberen.



Artiesten en hun albums: zie takenbundel

13 Statistieken (count, sum, min, max, average)

13.1 count

De Stream method count geeft je het aantal waarden in de Stream.

Je wijzigt de class Main:

```
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        System.out.println(Stream.of("sla", "wortel", "kool", "biet").count());
    }
}
```

Je kan dit uitproberen.

13.2 statistieken op int waarden

De interface IntStream is een naar performantie geoptimaliseerde Stream voor int waarden. Deze interface bevat de statistische methods count, sum, min, max en average.

13.2.1 sum

Je wijzigt de class Main:

```
import java.util.stream.IntStream;
class Main {
    public static void main(String[] args) {
        IntStream stream = IntStream.of(1, 3, 4, 7);
        System.out.println(stream.sum());
    }
}
```

❶
❷

- (1) Je maakt met de method of een IntStream met enkele waarden.
- (2) De method sum geeft je een int met de som van de waarden in de Stream.

Je kan dit uitproberen.

13.2.2 min, max en average

De method min geeft je de kleinste waarde in de IntStream.

Deze method geeft echter geen int terug omdat een IntStream ook leeg kan zijn.

De method min kan in dat geval geen realistische int waarde teruggeven.

De method min geeft een OptionalInt terug.

De class OptionalInt heeft dezelfde betekenis als een Optional<Integer>, maar is performanter.

De OptionalInt die de method min teruggeeft bevat 'niets' als de IntStream leeg is of bevat de kleinste waarde in de stroom als de stroom minstens één int bevat.

Je wijzigt de laatste opdracht in de class Main naar

```
stream.min().ifPresent(kleinste->System.out.println(kleinste));
```

Je kan dit uitproberen.

De method max geeft op dezelfde manier een OptionalInt terug.

De method average geeft op dezelfde manier een OptionalDouble terug.

13.2.3 Van een gewone Stream naar een IntStream

Je kan een Stream, die waarden bevat die geen int zijn, transformeren naar een IntStream met de Stream method `mapToInt`. Je kan daarna op die IntStream statistieken vragen.

Je wijzigt de class Main: je toont de som van alle letters in meerdere groenten:

```
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        System.out.println(
            Stream.of("sla", "wortel", "kool", "biet")
                .mapToInt(groente -> groente.length())
                .sum());
    }
}
```

❶

- (1) Je converteert de `Stream<String>` naar een `IntStream` met de method `mapToInt`. Je geeft aan de method `mapToInt` een lambda mee. De method `mapToInt` voert die lambda uit per `String` in de `Stream<String>` en geeft die waarde mee als parameter van de lambda. De lambda converteert die waarde naar een `int` en geeft die `int` terug.

Je kan dit uitproberen.



Kleinste oppervlakte: zie takenbundel

13.3 statistieken op long waarden

De interface `LongStream` is een specifieke Stream voor long waarden en werkt zoals een `IntStream`.

Je kan een Stream, die waarden bevat die geen long zijn, transformeren naar een `LongStream` met de Stream method `mapToLong`. Je kan daarna op die `LongStream` statistieken vragen.

13.4 statistieken op double waarden

De interface `DoubleStream` is een specifieke Stream voor double waarden en werkt zoals een `IntStream`.

Je kan een Stream, die waarden bevat die geen double zijn, transformeren naar een `DoubleStream` met de Stream method `mapToDouble`. Je kan daarna op die `DoubleStream` statistieken vragen.

13.5 min en max op waarden die geen int, long of double zijn

De Stream methods `min` en `max` geven de kleinste en grootste waarde van de waarden in een Stream. De types van de waarden in de Stream kunnen verschillen van `int`, `long` of `double`.

Je geeft aan de `min` en de `max` method een lambda mee die de functional interface `Comparator` implementeert. De `min` en `max` methods gebruiken intern die lambda om de kleinste of grootste waarde in de Stream te bepalen.

Je wijzigt de class Main: je toont het kleinste getal in een reeks `BigDecimal`s:

```
import java.math.BigDecimal;
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        Stream.of(
            BigDecimal.valueOf(1.1), BigDecimal.valueOf(0.9), BigDecimal.valueOf(0.5))
            .min((getal1, getal2) -> getal1.compareTo(getal2))
            .ifPresent(kleinste -> System.out.println(kleinste));
    }
}
```

❶

❷

- (1) Je geeft aan de method `min` een lambda mee die de interface `Comparator` implementeert.
- (2) De method `min` geeft een `Optional<BigDecimal>` terug.
Deze `Optional` bevat 'niets' als de `Stream` geen enkele waarde bevat.
Deze `Optional` bevat de kleinste waarde als de `Stream` wel waarden bevat.

Je kan dit uitproberen.



Laatste land: zie takenbundel

13.6 sum op waarden die geen int, long of double zijn

Als je de som wil maken van waarden die geen `int`, `long` of `double` zijn, gebruik je de `Stream` method `reduce`. Je leert deze method in het volgende hoofdstuk kennen.

14 Waarden in de Stream reduceren tot één waarde

Je leerde al de methods `count`, `sum`, `min`, `max` en `average` kennen.

Deze methods 'reduceren' de waarden in een `Stream` tot één waarde.

Voorbeeld: de `sum` method reduceert een `IntStream` met de waarden 4, 3 en 5 tot één waarde: 12

Je leert in dit hoofdstuk de `Stream` method `reduce` kennen.

Ook deze method reduceert de waarden in de `Stream` tot één waarde.

De method kan alle types waarden reduceren, niet enkel `int`, `long` en `double`.

De eerste versie van de `reduce` method (met twee parameters) is de meest algemene versie.

We leggen de betekenis van die parameters uit aan de hand van volgend reductie voorbeeld:

we reduceren een `Stream` met drie `BigDecimal` waarden (1.1, 0.9 en 0.5) tot hun som: 2.5

- de 1° parameter is de eindwaarde van de reductie als de stroom geen waarde bevat. Dit zal bij ons de waarde 0 zijn: de som van een stroom zonder waarden is 0.
- de 2° parameter is een lambda waarmee je de reductie stap per stap uitwerkt op basis van waarden in de `Stream`. De method `reduce` roept die lambda op per waarde in de `Stream`. Deze lambda is in ons voorbeeld (`vorigeSom, getal`) -> `vorigeSom.add(getal)` en

- krijgt de vorige waarde mee waarmee je de som opbouwde op basis van de waarden in de `Stream` die de method `reduce` reeds verwerkte.
- geeft de nieuwe waarde terug waarmee je de som verder opbouwt.

De method `reduce` itereert 3 keer, één per één over de `BigDecimal`s 1.1, 0.9 en 0.5

- iteratie 1: `vorigeSom` bevat de waarde in de 1° parameter van `reduce`: 0. (de waarde als de `Stream` geen waarden zou bevatten). Jij geeft als resultaat $0 + 1.1$ (dus 1.1) terug.
- iteratie 2: `vorigeSom` bevat 1.1: het resultaat van de lambda oproep in iteratie 1. Jij geeft als resultaat $1.1 + 0.9$ (dus 2.0) terug.
- iteratie 3: `vorigeSom` bevat 2.0: het resultaat van de lambda oproep in iteratie 2. Jij geeft als resultaat $2.0 + 0.5$ (dus 2.5) terug.

De `reduce` method geeft je het eindresultaat van de reductie (2.5) terug.

Je wijzigt de class `Main`: je maakt de som van enkele `BigDecimal`s:

```
import java.math.BigDecimal;
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        System.out.println(
            Stream.of(
                BigDecimal.valueOf(1.1), BigDecimal.valueOf(0.9), BigDecimal.valueOf(0.5))
                .reduce(BigDecimal.ZERO, (vorigeSom, getal) -> vorigeSom.add(getal)));
    }
}
```

Je kan dit uitproberen.

De tweede versie van de `reduce` method bevat één parameter. Je vult deze parameter met een lambda die dezelfde betekenis heeft als de lambda in de tweede parameter van de `reduce` method met twee parameters. We reduceren terug 1.1, 0.9 en 0.5 tot hun som: 2.5.

- iteratie 1: De `reduce` method onthoudt de huidige waarde (1.1) als tussenresultaat.
- iteratie 2: `vorigeSom` bevat het tussenresultaat van iteratie 1. Jij geeft als resultaat $1.1 + 0.9$ (dus 2.0) terug.
- iteratie 3: `vorigeSom` bevat het tussenresultaat van iteratie 2. Jij geeft als resultaat $2.0 + 0.5$ (dus 2.5) terug.

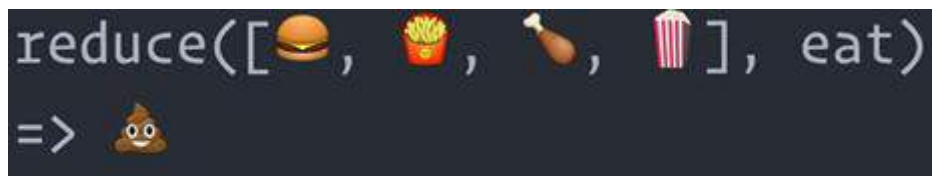
Deze versie van reduce geeft een `Optional<BigDecimal>` terug.
Deze bevat 'niets' als de Stream geen enkele waarde bevat.

Je wijzigt de class Main:

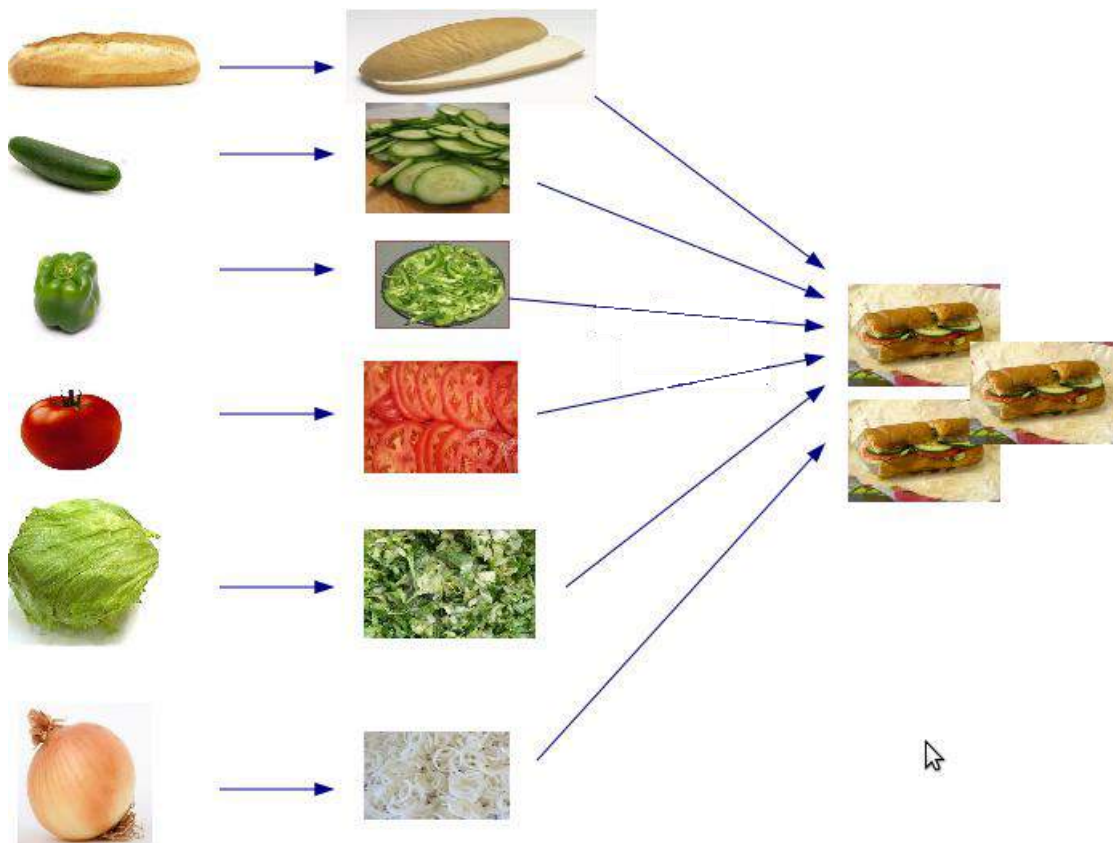
```
import java.math.BigDecimal;
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        Stream.of(
            BigDecimal.valueOf(1.1), BigDecimal.valueOf(0.9), BigDecimal.valueOf(0.5))
            .reduce((vorigeSom, getal) -> vorigeSom.add(getal))
            .ifPresent(som -> System.out.println(som));
    }
}
```

Je kan dit uitproberen.

Een grappige voorstelling van de method reduce die de vier elementen via het eet proces reduceert tot één element:



Een voorstelling van de combinatie van map (linkse pijlen) en reduce (rechtse pijlen)



15 Meerdere streams combineren tot één stream

Je combineert met de Stream method `flatMap` meerdere Streams tot één Stream.

Je combineert als eerste voorbeeld een Stream met de waarden Joe en Jack en een Stream met de waarden William en Averell tot één Stream met alle waarden uit de oorspronkelijke Streams: Joe, Jack, William en Averell.

Je wijzigt de class Main:

```
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        Stream.of(Stream.of("Joe", "Jack"), Stream.of("William", "Averell")) ❶
            .flatMap(stream -> stream) ❷
            .forEach(voornaam -> System.out.println(voornaam));
    }
}
```

- (1) Je maakt een Stream waarin elke waarde een Stream is.
- (2) Je geeft aan de method `flatMap` een lambda mee. De method `flatMap` roept die lambda op per waarde in de Stream. De parameter van die lambda is de waarde uit de Stream. Je lambda moet een Stream teruggeven (bij ons dezelfde als de Stream in de parameter). De method `flatMap` geeft je één Stream terug met alle String waarden uit returnwaarden (Streams) van de lambda oproepen.

Je kan dit uitproberen.

Je toont als tweede voorbeeld een gesorteerde lijst van de unieke woorden over meerdere boekentitels heen.

```
import java.util.Arrays;
import java.util.stream.Stream;
class Main {
    public static void main(String[] args) {
        Stream.of("The lord of the rings", "The hobbit")
            .map(titel -> titel.split(" ")) ❶
            .flatMap(array -> Arrays.stream(array)) ❷
            .map(woord -> woord.toLowerCase()) ❸
            .distinct()
            .sorted()
            .forEach(woord -> System.out.println(woord));
    }
}
```

- (1) De waarden in de oorspronkelijke Stream zijn Strings (de titels van boeken). Je maakt op basis van deze Stream een nieuwe Stream. De waarden in deze nieuwe Stream zijn String arrays. De eerste waarde is een array met de waarden The, lord, of, the en rings. De tweede waarde is een array met de waarden The en hobbit.
- (2) Je geeft aan de method `flatMap` een lambda mee. De method `flatMap` roept die lambda op per waarde in de Stream en geeft die waarde (een String array) mee als parameter van de lambda. Je lambda maakt een Stream bestaande uit de waarden in die String array. De method `flatMap` geeft je één Stream terug met alle String waarden uit returnwaarden (Streams) van de lambda oproepen.
- (3) Je zet alle woorden in de Stream om naar hun versie in kleine letters.

16 Parallel stream

De Stream methods gebruiken intern standaard één thread:
de thread waarop je de Stream methods oproept.

Als een Stream véél data bevat, kan je op die Stream de method `parallel` uitvoeren naast de andere Stream methods. Die andere methods gebruiken dan intern meerdere threads om sneller tot een eindresultaat te komen. Dit is voor jou een korte manier om aan multithreading te doen.

Als je op een Stream meerdere methods uitvoert, geeft het niet welke van die methods `parallel` is, als het maar niet de laatste is.



Opgepast, als de Stream weinig data bevat, kan met de method `parallel` het starten en beheren van de threads zoveel tijd kosten, dat de Stream methods trager werken dan zonder `parallel`! Je ziet dit in onderstaand voorbeeld.

Je wijzigt de class Main:

```
import java.math.BigDecimal;
import java.util.Random;
import java.util.stream.Stream;
class Main {
    private static void zonderParallel(long aantalWaarden) {
        Random random = new Random();
        long voor = System.nanoTime();
        Stream.generate(() -> BigDecimal.valueOf(random.nextDouble()))
            .limit(aantalWaarden)
            .filter(getal -> getal.remainder(BigDecimal.valueOf(2))
                .compareTo(BigDecimal.ZERO) == 0)
            .max((vorigGrootste, getal) -> vorigGrootste.compareTo(getal));
        System.out.println(String.format("%.16d: %.16d zonder parallel",
            aantalWaarden, System.nanoTime() - voor));
    }
    private static void metParallel(int aantalWaarden) {
        Random random = new Random();
        long voor = System.nanoTime();
        Stream.generate(() -> BigDecimal.valueOf(random.nextDouble()))
            .parallel()
            .limit(aantalWaarden)
            .filter(getal -> getal.remainder(BigDecimal.valueOf(2))
                .compareTo(BigDecimal.ZERO) == 0)
            .max((vorigGrootste, getal) -> vorigGrootste.compareTo(getal));
        System.out.println(String.format("%.16d: %.16d met parallel",
            aantalWaarden, System.nanoTime() - voor));
    }
    public static void main(String[] args) {
        Stream.of(10, 100, 1_000, 10_000, 100_000, 1_000_000, 10_000_000)
            .forEach(aantalWaarden -> {
                zonderParallel(aantalWaarden); metParallel(aantalWaarden);
            });
        System.out.println("the end");
    }
}
```

- (1) Deze method berekent de grootste waarde van enkel even (dus niet oneven) random BigDecimals zonder multithreading.
- (2) Je onthoudt het getal dat de systeemtijd voorstelt in nanoseconden.
- (3) Je maakt een oneindige Stream met random BigDecimals.
- (4) Je beperkt het aantal BigDecimals tot het aantal waarden in `aantalWaarden`.
- (5) Je houdt enkel de even getallen over.

- (6) Je bepaalt het grootste van deze even getallen.
 - (7) Je toont hoeveel nanoseconden het duurde om dit alles uit te voeren.
 - (8) Deze method berekent ook de grootste waarde van enkel random even BigDecimals, maar mét multithreading.
 - (9) Je laat de andere Stream methods multithreading gebruiken.
 - (10) Je maakt een reeks van enkele getallen en je roept met ieder van die getallen de methods `zonderParallel` en `metParallel` mee.
- Deze methods moeten zo iedere keer meer random BigDecimals verwerken.

Je kan dit uitproberen.

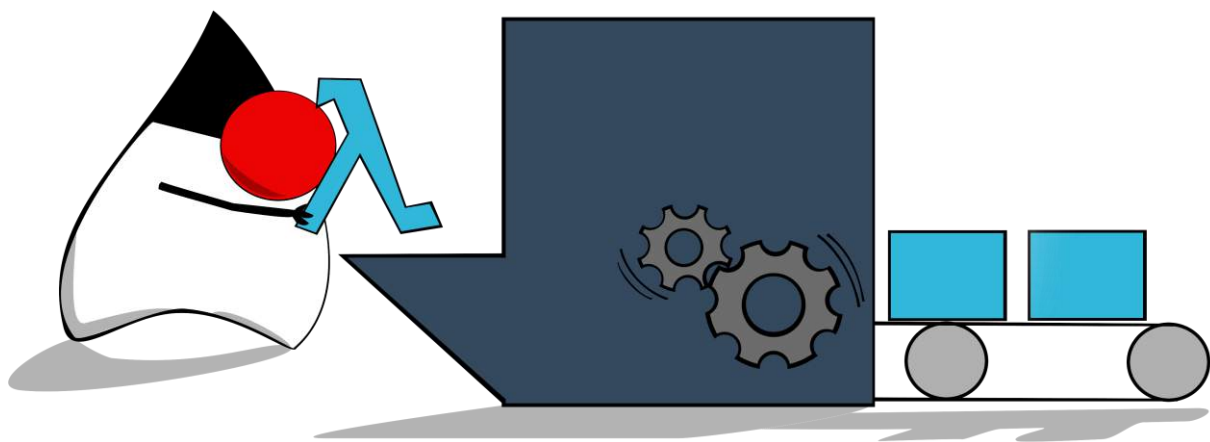
Bij kleine aantallen is de versie zonder `parallel` sneller dan die met `parallel`.

Hoe groter de aantallen, hoe meer de versie met `parallel` sneller is dan die zonder `parallel`.

Je hebt nu de mogelijkheden van een Stream gezien.

Je kan een Stream visualiseren als een machine die data verwerkt.

Jij stuurt die machine aan door lambda's aan die machine te voederen:



17 Method references

Een method reference is een alternatieve syntax om een lambda uit te drukken.

Een method reference is korter en (soms) leesbaarder dan de bijbehorende lambda.

Je kan een lambda vervangen door een method reference,

als de code van een lambda bestaat uit één opdracht die een method oproep is.

Er bestaan vier vormen van method references, waarvan je hieronder voorbeelden ziet:

Je wijzigt de class Main: je toont de omgekeerde versie van enkele Strings:

```
import java.util.stream.Stream;
```

```
class Main {
```

```
    private static String omgekeerd(StringBuilder builder) {
        return builder.reverse().toString();
    }
```

```
    public static void main(String[] args) {
```

```
        Stream.of("repeel", "lepel")
```

```
            .map(naam -> new StringBuilder(naam));
```

```
            .map(StringBuilder::new)
```

```
            .map(naam -> omgekeerd(naam))
```

```
            .map(Main::omgekeerd)
```

```
            .map(naam -> naam.toLowerCase());
```

```
            .map(String::toLowerCase)
```

```
            .forEach(naam -> System.out.println(naam));
```

```
            .forEach(System.out::println);
```

```
    }
```

```
}
```

①
②
③
④
⑤
⑥
⑦
⑧

- (1) Deze lambda maakt een StringBuilder op basis van een String met de StringBuilder constructor. Je vervangt deze lambda door de method reference bij (2).
- (2) Dit is de eerste vorm van een method reference: de oproep van een constructor. Je tikt de class waarvan je de constructor oproept, gevolgd door ::new. Java roept die constructor op en geeft de lambda parameter bij (1) mee als constructor parameter.
- (3) Deze lambda maakt een String op basis van de omgekeerde versie van een String. De lambda roept daarbij de static method omgekeerd van de class Main op. Je vervangt deze lambda door de method reference bij (4)
- (4) Dit is de tweede vorm van een method reference: de oproep van een static method. Je vermeldt een class, gevolgd door ::, gevolgd door een static method in die class. Java roept die method op en geeft de lambda parameter bij (3) mee als parameter. Java geeft de returnwaarde van deze method oproep terug aan de method map.
- (5) Deze lambda maakt een String op basis van de kleine letter versie van een String. Je vervangt deze lambda door de method reference bij (6)
- (6) Dit is de derde vorm van een method reference: de oproep van een method op een lambda parameter. Je tikt de class van de lambda parameter bij (5), gevolgd door ::, gevolgd door een method van die class. Java roept die method op die lambda parameter op en geeft de returnwaarde van die method terug aan de method map.
- (7) Deze lambda toont iedere naam. Je vervangt deze lambda door een method reference bij (8)
- (8) Dit is de vierde vorm van een method reference: de oproep van een method van een bepaald object. Je vermeldt dit object (System.out), gevolgd door ::, gevolgd door de naam van een method die je op dit object kan uitvoeren. Java roept die method op dit object op en geeft de lambda parameter bij (7) mee als parameter van deze method oproep.



Sterrenbeelden 2: zie takenbundel

18 Herhalingsoefeningen



Actrices tellen: zie takenbundel



Vicki: zie takenbundel

COLOFON

Domeinexpertisemanager	Jean Smits
Moduleverantwoordelijke	
Auteurs	Hans Desmet
Versie	7/10/2016
Codes	Peoplesoftcode: Wettelijk depot:

Omschrijving module-inhoud

Abstract	Doelgroep	Opleiding Java Ontwikkelaar
	Aanpak	Zelfstudie
	Doelstelling	Lambda's kunnen gebruiken
Trefwoorden		Lambda
Bronnen/meer info		