

7 Lambda's Streams

Programmeren 2 – Java

2017 - 2018

KdG Karel de Grote
Hogeschool

Kris Behiels

Jan De Rijke

Mark Goovaerts

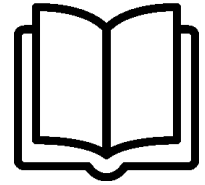
Programmeren 2 - Java

1. Herhaling en Collections
2. Generics en documenteren
3. Annotations en Reflection
4. Testen en logging
5. Design patterns (deel 1)
6. Design patterns (deel 2)

7. Lambda's en streams

8. Persistentie (JDBC)
9. XML en JSON
10. Threads
11. Synchronization
12. Concurrency





- E-book: "Java SE 8 Lambdas and Streams"
p.1 ev (Java How to Program, Tenth Edition)

Agenda

1. Lambda Expressions

- Inleiding
- Basisvormen
- Methodereferenties
- Building Blocks (`java.util.function`)



2. Streams

- Inleiding
- Creatie
- Method types
- Methoden: `forEach`, `map`, `filter`, `findFirst`, ...
- Parallel streams (later)
- Infinite streams



Lambda's

Inleiding



- JDK 8: Eindelijk **functioneel programmeren** in Java...
- Belangrijkste toevoeging sinds "Generics" (JDK 5)
- Ondertussen aanwezig in vrijwel alle moderne programmeertalen (ook JavaScript)
- Concept bestaat reeds zeer lang! → Lisp °1958

Inleiding

- In Objectgeoriënteerd programmeren draait alles rond het "Object".
- In Functioneel programmeren gaat het om de "Function". Je geeft geen waarde, maar een functie als parameter door (geen toestand, maar wel een gedrag)
- In Java kan je functioneel programmeren met behulp van "Lambda Expressions"
 - Code beter leesbaar
 - Minder schrijfwerk, vooral bij Collections
 - Parallele verwerking eenvoudiger (zie later)



Voordelen



• Oud:

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

• Nieuw:

```
button.setOnAction(event -> System.out.println("Je hebt geklikt"));
```

- Andere manier van denken, functioneel programmeren
- Code eenvoudiger te lezen en te onderhouden



Voordelen



- Je schrijft iets wat er als een functie uit ziet:
 - `Arrays.sort(testStrings,`
`(s1, s2) -> s1.length() - s2.length());`
 - `Collections.sort(piloten,`
`Comparator.comparing(p -> p.getNaam));`
 - `Timer.timerOperation(() -> sortArray(size));`
- Je krijgt een instantie van een klasse die de interface implementeert die op die plaats verwacht wordt.
 - De interface moet een zogenaamde "**functionele Interface**" zijn, ook "**Single Abstract Method**" genoemd.
 - Dit is een interface met exact één abstracte methode.
 - Opmerking: Vanaf Java 8 kan een interface ook concrete methoden hebben ("default methods") en ook static methoden.

Algemene syntax

- Vervang:

```
new SomeInterface() {  
    @Override  
    public SomeType someMethod(args) {  
        body  
    }  
}
```

- door:

```
(args) -> { body }
```

Java 1.0 interface implementatie

- We willen volgende `java.util.Arrays.sort` methode aanroepen:

```
public static <T> void sort(T[] a, Comparator<? super T> c)
```

- Volledige syntax: we implementeren de interface in de inner klasse `MyComparator` en maken een object aan:

```
Arrays.sort(testStrings, new MyComparator());  
  
class MyComparator implements Comparator <String>() {  
    @Override  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
});
```

- Als we de `Comparator` maar eenmaal aanmaken hoeven we hem geen naam te geven; we kunnen een **anonieme inner class** gebruiken (zie volgende slide)

Java 1.0 interface implementatie

- Volledige syntax (zonder anonieme inner klasse):

```
Arrays.sort(testStrings, new MyComparator());

class MyComparator implements Comparator <String> {
    @Override
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
});
```

- Java 1.0: anonieme inner klasse:

```
Arrays.sort(testStrings, new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
});
```

Van Java 1 naar Java 8

- Java 1.0: anonieme inner klasse:

```
Arrays.sort(testStrings, new Comparator<String>() {  
    @Override  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
});
```

- In Java 8 is er een kortere syntax voor volgend speciaal geval:
 - Je gebruikt de default constructor (= constructor van superklasse)
geen parameters: de compiler weet hoe hij het object moet aanmaken.
Je kan de constructor aanroep dus weglaten.
 - Je implementeert een interface met maar één methode
de compiler weet welke methode je wil overriden, want er is er maar één.
Je kan de methodenaam dus ook weglaten

Van Java 1 naar Java 8

- Java 1.0: anonieme inner klasse:

```
Arrays.sort(testStrings, new Comparator<String>() {  
    @Override  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
});
```

- Java 8 lambda expressie (let op het pijltje):

```
Arrays.sort(testStrings,  
    (String s1, String s2) -> {  
        return s1.length() - s2.length();  
    }  
);
```



Java 8 λ

- Java 8 lambda expressie:

```
Arrays.sort(testStrings,  
    (String s1, String s2) -> {  
        return s1.length() - s2.length();  
    }  
);
```

- Je implementeert een interface met maar één methode
=> De compiler kan de types van de functieparameters afleiden

```
Arrays.sort(testStrings,  
    (s1, s2) -> {  
        return s1.length() - s2.length();  
    }  
);
```

Korter



Java 8 λ

- Java 8 lambda expressie:

```
Arrays.sort(testStrings,  
            (s1, s2) -> { return s1.length() - s2.length(); } );
```

- Het kan nog korter als de functie maar één statement bevat
=> return en accolades mogen weg

Nog korter:

```
Arrays.sort(testStrings, (s1, s2) -> s1.length()-s2.length());
```



Samenvatting: van Java 1 naar Java 8

- We willen de `java.util.Arrays.sort` methode aanroepen:

```
public static <T> void sort(T[] a, Comparator<? super T> c)
```

- Volledige syntax:

```
Arrays.sort(testStrings, new MyComparator());  
  
class MyComparator implements Comparator <String>() {  
    @Override  
    public int compare(String s1, String s2) {  
        return s1.length() - s2.length();  
    }  
});
```

- Java 8 Lambda expression:

```
Arrays.sort(testStrings, (s1, s2) -> s1.length()-s2.length());
```

Speciaal geval: 1 parameter

- We willen volgende javafx Button methode aanroepen:

```
setOnAction(EventHandler<ActionEvent> value)
```

- Volledige syntax:

```
button.setOnAction(new MyButtonHandler);

class MyButtonHandler implements EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent e) {
        System.out.println("'t is gebeurd: " + e);
    }
}
```

- Java 8 Lambda expression:

```
button.setOnAction((e) -> System.out.println("'t is gebeurd: " + e));
```

- Nog korter (omdat er maar 1 parameter is: haakjes weg):

```
button.setOnAction(e -> System.out.println("'t is gebeurd: " + e));
```

Voorbeeld zonder parameters

- Oud:

```
Timer.timerOperation(new Operation() {  
    @Override  
    public void runOperation() {  
        sortArray(size);  
    }  
});
```

- Nieuw:

```
Timer.timerOperation(() -> sortArray(size));
```

Indien er geen parameters zijn: gebruik gewoon **ronde haakjes**.



Nieuwe annotatie

@functionalInterface

- De compiler controleert of de interface slechts één abstracte methode bevat (**Single Abstract Method** of **SAM**)
- Geeft aan dat de interface met lambda kan gebruikt worden
- Is net zoals de @Override annotatie niet verplicht!

```
@functionalInterface
public interface Operation {
    void runOperation();
}
```

Functioneel programmeren

- Lambda's doen eigenlijk hetzelfde als anonieme inner klassen
 - Je gebruikt ze voor zeer korte functie definities.
 - Er wordt slechts één instantie gemaakt.
- Lambda's voegen extra voorwaarden toe:
 - De default constructor van de superklasse wordt gebruikt
 - Er dient exact één methode geïmplementeerd te worden
 - Er zijn geen object attributen (heeft ook weinig zin als je maar één methode hebt)
- Geen data, één enkele methode: dit zijn eigenlijk functies
- De compacte syntax voelt ook aan als een functie die je doorgeeft.

Methode Referentie ::

- Nieuwe Java 8 operator ::
- Verwijzing naar een methode of constructor
 - zonder die direct uit te voeren
 - zonder parameters

Methode Ref Type	Voorbeeld
Class::staticMethod	Math::pow
Class::instanceMethod	String::compareTo
object::instanceMethod	System.out::println
Class::new	Piloot::new

Speciaal geval: methode referentie

Java 8 lambda expressie:

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

- Kortere syntax voor volgend speciaal geval:
 - De lambda expressie bevat enkel een aanroep van een bestaande methode (Object::method in het voorbeeld)
 - De parameters worden onveranderd volgens een conventie (zie later) doorgegeven

Java 8 lambda expressie met methode referentie:

```
button.setOnAction(System.out::println);
```

Speciaal geval: methode referentie

Java 8 lambda expressie met methode referentie:

```
button.setOnAction(System.out::println);
```

- Merk op dat de methode `println` niet onmiddellijk wordt aangeroepen. `System.out.println(event)` vormt de definitie van de inhoud van de eventhandler functie.
- Omdat de parameter `event` in de verkorte vorm onveranderd doorgegeven wordt aan de methode kan je dit dus **niet** gebruiken voor:

```
button.setOnAction(event -> System.out.println("Het is gebeurd: " + event));
```


Parameterconventies voor methode referenties

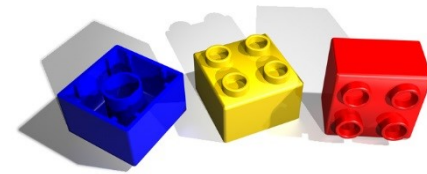
- Alle lambda parameters worden onveranderd, in volgorde, doorgegeven aan de methode referentie

Methode Ref Type	Voorbeeld	Equivalente Lambda
Class::staticMethod	Math::pow	(x, y) -> Math.pow(x, y)
Class::instanceMethod	String::compareTo	(x, y) -> x.compareTo(y)
object::instanceMethod	System.out::println	(x) -> System.out.println(x)
Class::new	Piloot::new	(x,y) -> new Piloot(x,y)

- Class::InstanceMethod heeft een licht afwijkende vorm:

KdG (x, y) -> x.compareTo(y)





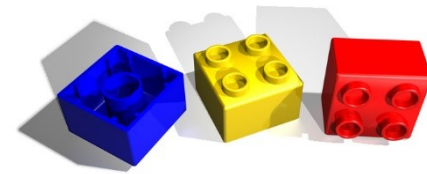
Building Blocks (java.util.function)

- De package `java.util.function` bevat heel wat herbruikbare **functionele interfaces**, je gebruikt ze alsof het functies waren.
- Interfaces met hun specifieke functies:

– Predicate <T>	<code>boolean test (T t)</code>
– Function <T,R>	<code>R apply (T t)</code>
– BiFunction <T,U,R>	<code>R apply (T t, U u)</code>
– Consumer <T>	<code>void accept (T t)</code>
– BiConsumer <T,U>	<code>void accept (T t, U u)</code>
– Supplier <T>	<code>T get ()</code>
– BinaryOperator <T>	<code>T apply (T t1, T t2)</code>
– UnaryOperator <T>	<code>T apply (T t)</code>

Demodata

```
public class Artikels {  
    final static private List<Artikel> artikels = Arrays.asList(  
        new Artikel(2, "Asus", "R556LA-XX1116H", 399),  
        new Artikel(6, "HP", "Pavilion 15-p268nb", 649),  
        new Artikel(7, "Asus", "EeeBook X205TA", 239),  
        new Artikel(1, "Lenovo", "IdeaPad G50-80", 549),  
        new Artikel(3, "Lenovo", "IdeaPad Z70-80", 499),  
        new Artikel(10, "Asus", "K555LJ-DM706T", 849),  
        new Artikel(9, "Lenovo", "IdeaPad S21e-20", 199),  
        new Artikel(5, "MSI", "GP72Qe-016BE", 1199),  
        new Artikel(4, "Toshiba", "Satelite Pro R50-B-109", 399),  
        new Artikel(8, "Toshiba", "Satelite L50D-B-1CE", 649)  
    );  
    public static List<Artikel> getArtikels() {  
        return new ArrayList<>(artikels);  
    }  
}
```



Predicate functionele interface

boolean test(T t)

- "functie" om een conditie te testen
- Voorbeeld: Zoek in een collection naar elementen die aan de testconditie voldoen
- Voorbeeld gebruik (zoek eerste overeenkomst):

```
public static <T> T firstMatch(List<T> candidates,  
                                Predicate<T> matchFunction) {  
    for (T match : candidates) {  
        if (matchFunction.test(match)) {  
            return match;  
        }  
    }  
    return null;  
}
```


Predicate functionele interface: gebruik

- Toepassing met lambda expression:

```
List<Artikel> lijst = Artikels.getArtikels();

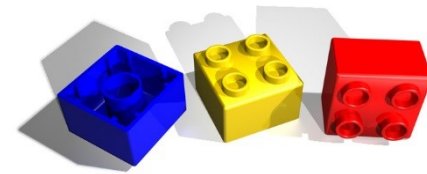
System.out.println(firstMatch(lijst, a -> a.getPrijs() > 500));
System.out.println(firstMatch(lijst,
                               a -> a.getMerk().equals("Lenovo")));
System.out.println(firstMatch(lijst, a -> a.getNummer() == 4));
```

Vroegere notatie
zonder lambda's



```
System.out.println(firstMatch(lijst,
                               new Predicate<Artikel>() {
                                   @Override
                                   public boolean test(Artikel a) {
                                       return a.getPrijs() > 500;
                                   }
                               }));
```





Function functionele interface

R **apply** (T t)

- "functie" die een **T** als invoer en een **R** als uitvoer heeft
- Voor transformatie van een waarde of een collection van waarden
- **BiFunction** is zelfde, maar **apply** heeft dan 2 argumenten
- Voorbeeld gebruik: (function zet elke element om naar double, waarna de som berekend wordt)

```
public static <T> double mapSum(List<T> entries,  
                                Function<T, Double> mapper) {  
    double sum = 0;  
    for (T entry : entries) {  
        sum += mapper.apply(entry);  
    }  
    return sum;  
}
```

Function functionele interface

- Toepassing met lambda expression:

```
List<Artikel> lijst = Artikels.getArtikels();  
  
System.out.println("Som artikelnummers: " +  
                    mapSum(lijst, a -> (double)a.getNummer()));  
System.out.println("Som prijzen: " +  
                    mapSum(lijst, a -> a.getPrijs()));  
System.out.println("Som prijzen: " +  
                    mapSum(lijst, Artikel::getPrijs));
```

Zelfde resultaat!

Lambda expressie kan vervangen worden door **method reference**, omdat de returnwaarde van `getPrijs` voldoet aan de type-declaratie in `Function` (vorige slide)

Function functionele interface: voorbeeld

- In een **comparator** passen we vaak dezelfde functie toe op de twee elementen die vergeleken worden.
- Voorbeeld: van de twee piloten die vergeleken worden halen we de naam, en dan doen we een compareTo van beide namen
- Klassieke code

```
Collections.sort(piloten, new Comparator<Piloot>() {  
    @Override  
    public int compare(Piloot p1, Piloot p2) {  
        return p1.getNaam().compareTo(p2.getNaam());  
    }  
});
```

- `Comparator.compareTo(Function f)` neemt als input de toe te passen functie (`p -> p.getNaam()`) en genereert zo'n Comparator voor je

Function functionele interface: voorbeeld

- Zelfde voorbeeld met Function interface:

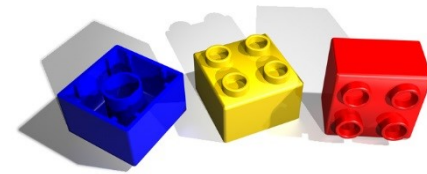
```
Collections.sort(piloten, Comparator.comparing(p -> p.getNaam()));
```

- Implementatie van **Comparator.comparing** in JDK source (vereenvoudigd):

```
public static <T> Comparator comparing(  
    Function<T,U extends Comparable> getKey){  
    return (Comparator<T>) (c1, c2) ->  
        getKey.apply(c1).compareTo(getKey.apply(c2));  
}
```

- **thenComparing**(Function f) laat je toe op extra attributen te sorteren
- In Comparator zijn een heel wat nieuwe static utility methoden: **reverseOrder**, **nullsFirst**...





Consumer functionele interface

`void accept(T t)`

- "functie" die een **T** als invoer heeft en iets met die **T** doet zonder iets terug te geven
- Om op een collection een bepaalde bewerking te doen (bijvoorbeeld alle elementen met een waarde verhogen)
- Voorbeeld gebruik:
 - default methode **forEach** in de `Iterable<T>` interface

Consumer functionele interface

- Toepassing met lambda expression:

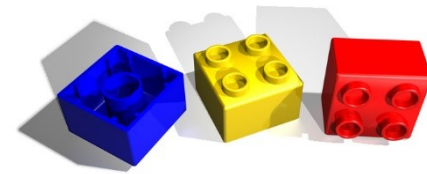
```
List<Artikel> lijst = Artikels.getArtikels();

Consumer<Artikel> print = a -> System.out.println(a);
lijst.forEach(print);

System.out.println("\nToekennen korting 10%:\n");
lijst.forEach(a -> a.setPrijs(a.getPrijs() * 0.9));
lijst.forEach(System.out::println);
```

OF lambda expression:
lijst.forEach(a -> System.out.println(a));





Supplier functionele interface

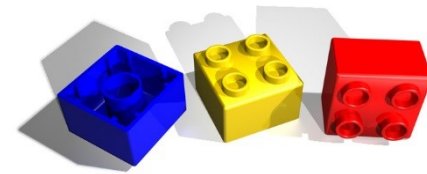
- **T** `get()`
 - "functie" zonder invoer die een **T** teruggeeft
- Voorbeelden gebruik:
 - Een nieuw object maken
 - Om het resultaat van een **static** methode op te halen
 - Als argument bij een **Stream** (zie later)

Supplier functionele interface

- In `java.util.logging` kan je als boodschap een `Supplier<String>` meegeven:

```
String user = "Carolus Magnus";  
String company = "KDG";  
logger.fine(() -> "Lambda concatenatie voor %s van %s " ,  
             user , company );
```

- Je kan dynamisch een boodschap samenstellen met de `lambda`
- De `lambda` wordt pas aangeroepen binnen de `log` methode na een check of het `log level` actief is (efficiënt).




UnaryOperator functionele interface

- **T** **apply** (T)
 - "functie" die een parameter **T** als invoer en eenzelfde type **T** (met een andere waarde) als uitvoer heeft, is een specialisatie van **Function**
- Voorbeeld gebruik:
 - In de **replaceAll** methode van de **List** interface (default methode)

UnaryOperator functionele interface

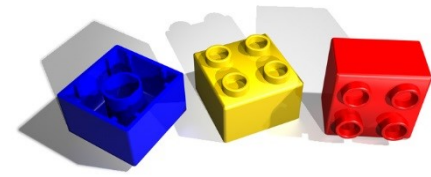
- Toepassing met lambda expression:

```
List<Artikel> artikels = Artikels.getArtikels();  
// replaceAll heeft parameter van het type UnaryOperator<T> operator  
  
artikels.replaceAll(a -> new Artikel(a.getNummer(), a.getMerk(),  
                                     a.getType(), a.getPrijs() * 1.25));  
artikels.forEach(System.out::println);
```



Vervangt elk artikel door een kopie met prijs die 25% hoger ligt

BinaryOperator functionele interface



- **T** **apply** (T t1, T t2)
 - "functie" die twee T's als invoer heeft en T teruggeeft, is een specialisatie van **BiFunction**<T,U,R> waar T, U en R hetzelfde type hebben
- Voorbeeld gebruik:
 - In de **reduce** methode van de **Stream**<T> interface (zie later)

Opdrachten

- Groeiproject

- module 6

- (deel 1 en 2: "Lambda's en method references")



- Opdrachten op BB

- ➔ Opgave Robomail

- MyFunction

- MySupplier



Agenda

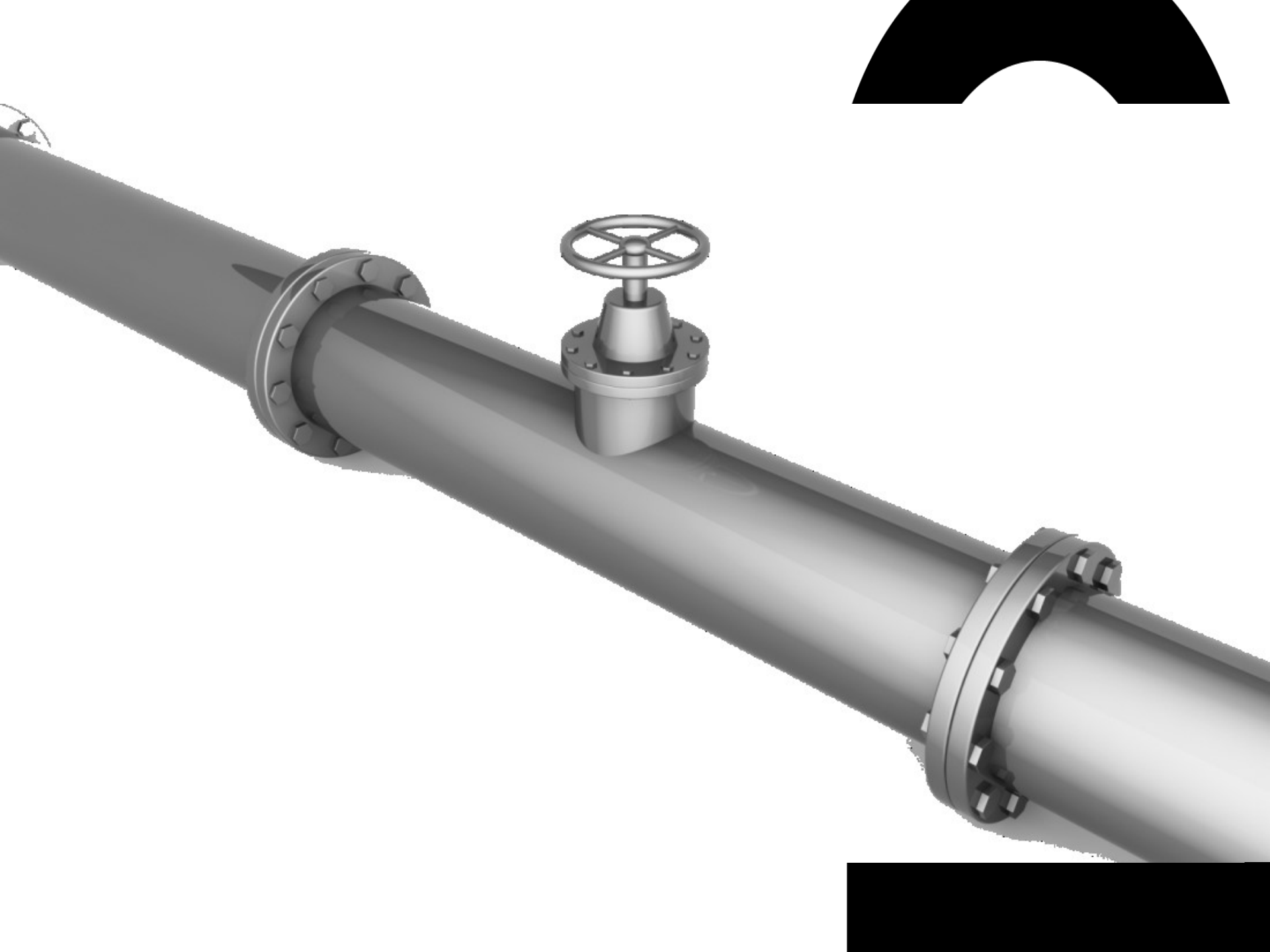
1. Lambda Expressions

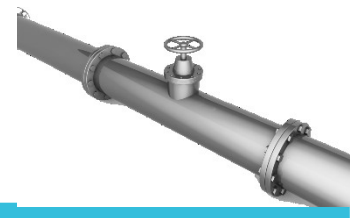
- Inleiding
- Basisvormen
- Methodereferenties
- Building Blocks (`java.util.function`)



2. Streams

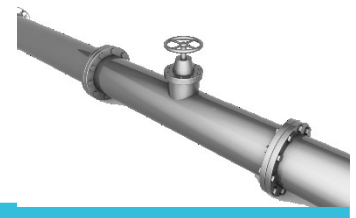
- Inleiding
- Creatie
- Method types
- Methoden: `forEach`, `map`, `filter`, `findFirst`, ...
- Parallel streams (later)
- Infinite streams





- **Streams:**

- Reeks elementen die een ketting van operaties doorlopen
- Hebben meer methoden dan Lists
 - `forEach`, `filter`, `map`, `sorted`, ...
- Zijn efficiënter!
 - Lazy evaluation, Automatic parallelization, Infinite streams
- Slaan geen gegevens op, zijn *wrappers* rond bestaande data
- Niet te verwarren met I/O Streams!



- **Streams:**

- Zijn geen data structuren, maar een "**pijplijn**" van operaties
- Alternatief voor iteraties (loops)
 - geen controle instructies (for, if, while, break...)
 - geen lokale variabelen, geen index
 - brengen *geen wijzigingen* aan in de onderliggende List of array
- Aanvaarden *lambda*'s als parameters
- Kunnen naar List of array omgezet worden

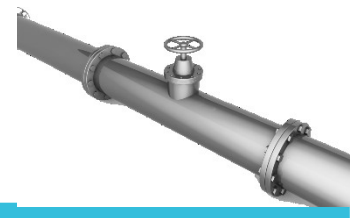
```
swimmers.stream()
```

```
.filter(...)
```

```
.map(...)
```

```
.collect(...);
```





- Java 8 Stream interface
 - `java.util.stream.Stream<T>`
 - Specialisaties voor primitieve datatypes:
 - `IntStream`
 - `LongStream`
 - `DoubleStream`

Demodata

```
public class Artikels {  
    final static private List<Artikel> artikels = Arrays.asList(  
        new Artikel(2, "Asus", "R556LA-XX1116H", 399),  
        new Artikel(6, "HP", "Pavilion 15-p268nb", 649),  
        new Artikel(7, "Asus", "EeeBook X205TA", 239),  
        new Artikel(1, "Lenovo", "IdeaPad G50-80", 549),  
        new Artikel(3, "Lenovo", "IdeaPad Z70-80", 499),  
        new Artikel(10, "Asus", "K555LJ-DM706T", 849),  
        new Artikel(9, "Lenovo", "IdeaPad S21e-20", 199),  
        new Artikel(5, "MSI", "GP72Qe-016BE", 1199),  
        new Artikel(4, "Toshiba", "Satelite Pro R50-B-109", 399),  
        new Artikel(8, "Toshiba", "Satelite L50D-B-1CE", 649)  
    );  
    public static List<Artikel> getArtikels() {  
        return new ArrayList<>(artikels);  
    }  
}
```


Werken met streams

1. Maak een stream (zie verder)
2. Specificeer 0 of meer "tussenliggende operaties"
 - Operatie op een stream
 - Resultaat is een nieuwe Stream (geeft data door naar volgende stap)
3. Specificeer een "eindoperatie"
 - Met deze operaties "consumeer" je de stream, m.a.w. je zet de stream om naar een eind object (bv. `forEach`, `collect`, ...)



Een voorbeeld

Tel het aantal artikels goedkoper dan €400

```
List<Artikel> artikels = Artikels.getArtikels();
```

- Iteratief:

```
int aantal = 0;
for (Artikel artikel : artikels) {
    if (artikel.getPrijs() > 400.0) {
        aantal++;
    }
}
System.out.println(aantal);
```

- Met Stream:

```
long aantalArtikels = artikels.stream()
    .filter(a -> a.getPrijs() > 400.0)
    .count();
System.out.println(aantalArtikels);
```

1. Maak Stream

2. Tussenliggende operatie

3. Eind operatie



Nog een voorbeeld

Tel de prijzen van alle artikels samen

```
List<Artikel> artikels = Artikels.getArtikels();
```

```
double totaal = artikels.stream()  
    .map(a -> a.getPrijs())  
    .reduce((a,b) -> a + b)  
    .get();  
System.out.println(totaal);
```

1. Maak Stream

2. Tussengeschiedte operatie

3. Eind operatie

- Reduce zorgt ervoor dat alle prijzen bij elkaar opgeteld worden → resultaat is een `Optional<Double>`, we vragen de waarde ervan op met `get()`



- Vanuit een List:

```
List<T> list = ...;  
list.stream()  
    .intermediate  
    .intermediate  
    ...  
    .terminal
```

- Vanuit een array:

```
T[] array= ... ;  
Stream.of(array)  
    .intermediate  
    .intermediate  
    ...  
    .terminal
```

Tussenliggende operaties retourneren een Stream, waarop dan de volgende method kan toegepast worden (**function chaining**)

Creatie (voorbeelden)

```
List<String> rijders = Arrays.asList(  
    "Lewis", "Nico", "Sebastian", "Kimi"  
);  
  
rijders.stream().forEach(System.out::print);
```

```
Stream<String> piloten = Stream.of(  
    "Lewis", "Nico", "Sebastian", "Kimi"  
);  
  
piloten.forEach(System.out::println);
```

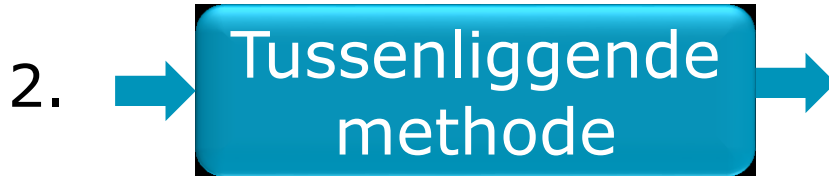
```
String[] drivers = {  
    "Lewis", "Nico", "Sebastian", "Kimi"  
};  
  
Stream.of(drivers).forEach(System.out::println);
```



Creatie: streams samenvoegen

```
Stream<String> piloten = Stream.concat(  
    Stream.of("Lewis", "Nico"),  
    Stream.of("Sebastian", "Kimi")  
);  
  
piloten.forEach(System.out::println);
```

Verdere Stream methode types



– `map` (en afgeleiden), `filter`, `distinct`, `sorted`, `peek`, `limit`, `skip`, `parallel`, `sequential`, `unordered`, ...



– `forEach`, `forEachOrdered`, `toArray`, `reduce`, `collect`, `min`, `max`, `count`, `anyMatch`, `noneMatch`, `findFirst`, `findAny`, `iterator`

- een eind methode verwerkt een stream van waarden. Het resultaat kan een enkele waarde zijn.

forEach

- Eenvoudige manier om over alle elementen van een **Stream** te itereren
- Gebruikt een functie (**Consumer**) die voor elk element van de **Stream** wordt uitgevoerd

```
Stream.of("One", "Two", "Three", "Four")
    .forEach(System.out::println);

IntStream.of(1, 2, 3, 4)
    .forEach(e -> System.out.println(e + 1));
```

Kan hier niet vervangen worden door method
reference `System.out::println`
Zie je ook waarom?

- **Voordelen:**

- Ontwikkeld voor lambda's
- Herbruikbaar
- Kan eenvoudig parallel uitgevoerd worden

- Wat kan **niet**?

- De iteratie tweemaal doen, **forEach** is een *eindoperatie*.
- Wijzigen welke elementen in de stream voorkomen. Dit kan wel via **map** en **reduce**.
DoubleStream heeft een ingebouwde "sum" methode
- De lus voortijdig *verlaten*

- Vergelijking met for:

```
List<Artikel> artikels = Artikels.getArtikels();
```

```
/* for */
```

```
for (Artikel artikel : artikels) {  
    artikel.setPrijs(artikel.getPrijs() - 50);  
    System.out.println(artikel);  
}
```

```
/* forEach */
```

```
artikels.stream().forEach(e -> e.setPrijs(e.getPrijs() - 50));  
artikels.stream().forEach(System.out::println);
```



forEach bestaat ook op Collection, dus in dit
specifieke geval mag je stream() weglaten!

- Hergebruik:

```
List<Artikel> artikels = Artikels.getArtikels();  
List<Artikel> lijst = Arrays.asList(  
    new Artikel(1, "Volvo", "S60", 35240),  
    new Artikel(2, "Audi", "A4", 32800)  
);  
  
Consumer<Artikel> korting = e -> e.setPrijs(e.getPrijs() - 50);  
  
artikels.stream().forEach(korting);  
artikels.stream().forEach(System.out::println);  
  
System.out.println();  
lijst.stream().forEach(korting);  
lijst.stream().forEach(System.out::println);
```

Consumer-object (λ implementatie functionele interface) wordt herbruikt

- Maakt een nieuwe stream die het resultaat is van het toepassen van een functie (**Function**) op **elk** element van de oorspronkelijke stream:

```
List<Integer> getallen = Arrays.asList( 1, 2, 3, 4, 5);  
getallen  
    .stream()  
    .map(i -> i * i)  
    .forEach(e -> System.out.print(e + " "));
```

- Afgeleide methoden:
 - **mapToInt** → maakt IntStream
 - **mapToDouble** → maakt DoubleStream
 - **flatMap** → combineert geneste Streams tot een enkelvoudige stream

map

Tussenliggende methode

```
List<Artikel> artikels = Artikels.getArtikels();  
// Verhoog met 21% BTW en Sorteer volgens nummer  
artikels  
    .stream()  
    .map(e -> new Artikel(e.getNummer(),  
                           e.getMerk(), e.getType(), e.getPrijs() * 1.21))  
    .sorted() // Volgens compareTo van klasse Artikel  
    .forEach(System.out::println);  
// Verhoog met 21% BTW en Sorteer volgens prijs  
artikels  
    .stream()  
    .map(e -> new Artikel(e.getNummer(), e.getMerk(),  
                           e.getType(), e.getPrijs() * 1.21))  
    .sorted((a, b) -> Double.compare(a.getPrijs(), b.getPrijs()))  
    .forEach(System.out::println);
```



- Maakt een nieuwe stream die alleen de elementen van de oorspronkelijke stream bevat die aan een bepaalde voorwaarde voldoen (**Predicate**)

```
// Maak een tabel met alle getallen van 1 tot en met 100
Integer[] getallenTabel = Stream
    .iterate(1, n -> n + 1)
    .limit(100)
    .toArray(Integer[]::new);

// Maak een stream met alle even getallen kleiner dan 40
Stream<Integer> getallen = Arrays.stream(getallenTabel)
    .filter(n -> n % 2 == 0)
    .filter(n -> n < 40);
getallen.forEach(a -> System.out.print(a + " "));
System.out.println();
```

```
List<Artikel> artikels = Artikels.getArtikels();

// Geef alle artikels van het merk Asus die minder dan €500 kosten
artikels
    .stream()
    .filter(e -> e.getPrijs() < 500)
    .filter(e -> e.getMerk().equals("Asus"))
    .forEach(System.out::println);

// Geef alle artikels van het merk Asus (oplopende prijs)
List<Artikel> asusLijst = artikels
    .stream()
    .filter(e -> e.getMerk().equals("Asus"))
    .sorted((a, b) -> Double.compare(a.getPrijs(), b.getPrijs()))
    .collect(Collectors.toList());

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



findFirst



- Geeft een **Optional** terug (zie verder)
- Mogelijk gebruik:
 - `Stream.findFirst().get()`
 - `Stream.findFirst().orElse(other object)`

```
// Maak een tabel met de getallen van 1 tot en met 100
Integer[] getallenTabel = Stream
    .iterate(1, n -> n + 1)
    .limit(100)
    .toArray(Integer[]::new);
Collections.shuffle(Arrays.asList(getallenTabel));
// Zoek in getallenTabel het eerste even getal kleiner dan 40
Optional<Integer> eerste = Stream.of(getallenTabel)
    .filter(n -> n % 2 == 0)
    .filter(n -> n < 40)
    .findFirst();
System.out.println(eerste.get());
```


findFirst



Eind methode

```
List<Artikel> artikels = Artikels.getArtikels();

// Zoek het eerste artikel van Lenovo goedkoper dan €500
Optional<Artikel> artikel = artikels
    .stream()
    .filter(e -> e.getPrijs() < 500)
    .filter(e -> e.getMerk().equals("Lenovo"))
    .findFirst();

if (artikel.isPresent()) {
    System.out.println(artikel.get());
} else {
    System.out.println("Geen artikel gevonden!");
}
```



Optional

- De waarde die `findFirst` teruggeeft is een element van de klasse **Optional**
- Declaratie: `java.util.Optional<T>`
- Wordt gebruikt om `null` te vervangen door een "niet aanwezige" waarde. Bevat allerlei methoden om te controleren op al dan niet aanwezige waarden (in plaats van op `null` te testen). Zie Javadoc.
- Enkele methoden:
 - `T get()` → geeft waarde terug indien aanwezig
 - `boolean isPresent()` → `true` als waarde aanwezig

Optional vloeiende stijl

- Enkele Optional methoden voor vloeiende (functionele) stijl
 - `ifPresent(Consumer)` : doe operatie op waarde van optional indien aanwezig
 - `orElse(U)` : geef waarde van optional of, indien niet aanwezig, waarde U
- Java 9
 - `ifPresentOrElse(Consumer c, Runnable r)` : doe operatie c op waarde van optional indien aanwezig, doe anders operatie r zonder waarde
 - een runnable is een functie zonder parameters (zie latere lesmodule: Concurrency)

findFirstFluid (Java 9)



```
List<Artikel> artikels = Artikels.getArtikels();

// Zoek het eerste artikel van Lenovo goedkoper dan €500
Optional<Artikel> artikel = artikels
    .stream()
    .filter(e -> e.getPrijs() < 500)
    .filter(e -> e.getMerk().equals("Lenovo"))
    .findFirst()
    .orElse(
        e -> System.out.println(e),
        () -> System.out.println("Geen artikel gevonden!");
    );
}
```



- Terminal operations **min** en **max**
 - **min**: geeft het minimale element terug in de vorm `Optional<T>` volgens een `Comparator`
 - **max**: idem, voor het maximale element

```
List<Artikel> artikels = Artikels.getArtikels();

// Druk het goedkoopste artikel af
Artikel goedkoopste = artikels
    .stream()
    .min((a, b) -> Double.compare(a.getPrijs(), b.getPrijs()))
    .get();
System.out.println(goodkoopste);

// Druk het duurste artikel af indien er een artikel aanwezig is
artikels.stream()
    .max((a, b) -> Double.compare(a.getPrijs(), b.getPrijs()))
    .ifPresent(System.out::println);
```



- Intermediate operation **limit**
- **Stream<T> limit(long maxSize)**
- Geeft een stream terug met maximaal **maxSize** elementen

```
// Maak een tabel met 10000 willekeurige getallen van 1 tot en met 1000
IntStream getallen = new Random().ints(10000, 1, 1001);
// Druk de eerste 10 gegenereerde getallen kleiner dan 100 af
getallen
    .filter(e -> e < 100)
    .limit(10)
    .forEach(e -> System.out.print(e + " "));
System.out.println();

// Druk de eerste 12 oneven getallen af
Stream.iterate(1, n -> n + 2)
    .limit(12)
    .forEach(n -> System.out.print(n + " "));
```



limit

```
// Neem de eerste 4 strings,  
// sorteer ze en voeg ze in één string samen  
List<String> piloten = Arrays.asList("Lewis", "Niko", "Sebastian",  
    "Kimi", "Valtteri", "Felipe");  
String result = piloten  
    .stream()  
    .limit(4)  
    .sorted()  
    .collect(Collectors.joining(", ")); // zie verder  
System.out.println(result);
```

- Terminal operation **collect**
- Om uit een **Stream** verschillende soorten objecten te maken

```
import static java.util.stream.Collectors  
gedaan zodat bijvoorbeeld Collectors.toList()  
afgekort wordt tot toList()...
```

–List

- `eenStream.collect(toList())`

–String

- `eenStream.collect(joining(delimiter))`

–Set

- `eenStream.collect(toSet())`

–Map

- `eenStream.collect(partitioningBy(...))`

- `eenStream.collect(groupingBy(...))`

Collectors.toList en toSet

```
List<Integer> nummers = Arrays.asList(2, 6, 8);
```

```
List<Artikel> specials =  
    nummers.stream()  
        .map(Artikels::zoekArtikel)  
        .collect(Collectors.toList());
```

static methode `zoekArtikel` in
klasse `Artikels`
De artikelnummers 2, 6 en 8
worden opgezocht.

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

```
List<String> piloten = Arrays.asList("Niko", "Lewis", "Kimi",  
                                     "Sebastian", "Valtteri", "Felipe");
```

```
Set<String> drivers = piloten.stream()  
    .filter(e -> e.charAt(0) < 'S')  
    .collect(Collectors.toSet());
```

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



Collectors.joining

```
List<Integer> nummers = Arrays.asList(1, 3, 9, 4, 8);

String types = nummers.stream()
    .map(Artikels::zoekArtikel)
    .map(Artikel::getType)
    .collect(Collectors.joining(", "));

System.out.println(types);
```

voegt artikel**types** in één string bijeen, gescheiden door komma's

Collectors.toCollection

- **Collectors.toCollection** heeft een nieuwe lege `Supplier<Collection>` als argument. Elk element van de stream wordt aan de collection toegevoegd

–ArrayList

- `eenStream.collect(toCollection(ArrayList::new))`

–TreeSet

- `eenStream.collect(toCollection(TreeSet::new))`

–Stack

- `eenStream.collect(toCollection(Stack::new))`

–Vector

- `eenStream.collect(toCollection(Vector::new))`

```
import static java.util.stream.Collectors gedaan zodat  
Collectors.toCollection()afgekort wordt tot toCollection()...
```

Collectors.toCollection

```
List<String> piloten = Arrays.asList("Lewis", "Niko", "Sebastian",  
                                     "Kimi", "Valtteri", "Felipe");  
  
TreeSet<String> drivers = piloten.stream()  
    .filter(e -> e.charAt(0) > 'K')  
    .collect(Collectors.toCollection(TreeSet::new));  
  
drivers.forEach(System.out::println);
```



Collectors.partitioningBy

- **partitioningBy**: Creatie Maps
 - Je moet een **predicate** (logische λ expressie) voorzien
 - Er wordt een **map** gemaakt met 2 lijsten, een met alle elementen die voldoen aan de predicate en een lijst met alle andere elementen.
 - Algemene vorm: **Map<Boolean, List<T>>**

Collectors.partitioningBy

```
List<Artikel> artikels = Artikels.getArtikels();

Map<Boolean, List<Artikel>> map = artikels
    .stream()
    .sorted()
    .collect(Collectors.partitioningBy(
        e -> e.getNummer() % 2 == 0));

List<Artikel> even = map.get(true);
List<Artikel> onEven = map.get(false);
```



Collectors.groupingBy

- **groupingBy**: Creatie Maps

- Je moet een **function** voorzien die voor elk element een key geeft
 - Het element wordt opgeslagen in een lijst bij die key
- Algemene vorm: **Map<T, List<R>>**

Collectors.groupingBy

• groupingBy

```
List<Artikel> artikels = Artikels.getArtikels();

Map<String, List<Artikel>> map = artikels
    .stream()
    .collect(Collectors.groupingBy(Artikel::getMerk));

map.forEach((k, v) -> System.out.printf("%-8s %s\n",
    k, v.stream()
        .map(e -> e.getType() + " -> €" + e.getPrijs())
        .collect(Collectors.joining(", ")))));
```

```
Lenovo    IdeaPad G50-80 -> €549.0, IdeaPad Z70-80 -> €499.0, IdeaPad S21e-20 -> €199.0
Toshiba   Satelite Pro R50-B-109 -> €399.0, Satelite L50D-B-1CE -> €649.0
MSI       GP72Qe-016BE -> €1199.0
HP        Pavilion 15-p268nb -> €649.0
Asus      R556LA-XX1116H -> €399.0, EeeBook X205TA -> €239.0, K555LJ-DM706T -> €849.0
```



Parallele Streams

- Een groot voordeel van Streams ten opzichte van loops is dat je ze makkelijk **parallel** kan verwerken
- Zie latere lesmodule ("Concurrency")

Infinite Streams

- **Stream.generate** (*valueGenerator*)
- **Stream.iterate** (*initialValue*,
valueTransformer)
- Gebruik:
 - Indien je een "on the fly" stream nodig hebt, zonder vaste grootte
 - De waarden worden niet berekend tot ze nodig zijn (lazy evaluation)
 - Om het process af te breken moet je uiteindelijk een "size limiting operation" zoals **limit** of **findFirst** uitvoeren.

Stream.generate

- **Stream.generate** (*valueGenerator*)
 - *valueGenerator* is een **Supplier** functionele interface
 - De functie wordt uitgevoerd als de stream elementen nodig heeft
 - De functie houdt de toestand bij zodat nieuwe waarden gegenereerd worden op basis van om het even welke voorgaande waarden

Stream.generate

- Stateless voorbeeld:

```
Random random = new Random();  
Supplier<Integer> generator = () -> random.nextInt(900) + 100;  
  
List<Integer> getallen = Stream.generate(generator)  
    .limit(100)  
    .sorted()  
    .collect(Collectors.toList());
```

Maakt een gesorteerde List van 100 willekeurige getallen in het bereik 100 .. 999



Stream.generate

- Statefull voorbeeld (helper class)

```
public class FibonacciMaker implements Supplier<Long> {  
    private long vorige = 0;  
    private long huidige = 1;  
  
    @Override  
    public Long get() {  
        long volgende = huidige + vorige;  
        vorige = huidige;  
        huidige = volgende;  
        return vorige;  
    }  
}
```

Stream.generate

- Statefull voorbeeld

```
public class Fibonacci {  
    public static void main(String[] args) {  
        Stream<Long> fibonacciStream =  
            Stream.generate(new FibonacciMaker());  
  
        List<Long> fibonacciGetallen = fibonacciStream  
            .limit(25)  
            .collect(Collectors.toList());  
  
        fibonacciGetallen.forEach(System.out::println);  
    }  
}
```

Stream.iterate

- **Stream.iterate**(*seed*, *valueTransformer*)
 - Laat je toe om een *seed* en een `UnaryOperator` *valueTransformer* te specificëren.
 - Het *seed* wordt het eerste element van de stream, *valueTransformer*(*seed*) wordt het tweede element, *valueTransformer*(tweede) het derde enzovoort..
 - Geeft niet noodzakelijk dezelfde resultaten bij parallelle uitvoering

Stream.iterate

- **iterate** voorbeelden

```
List<Integer> machten = Stream.iterate(1, n -> n * 2)
    .limit(21)
    .collect(Collectors.toList());

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

```
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
...
```

```
Stream.iterate(3, n -> n + 2)
    .skip(100)
    .limit(5)
    .forEach(System.out::println);
```

```
203
205
207
209
211
```



Reduction



- **T reduce**(T *identity*, BinaryOperator<t> *accumulator*)
 - Deze methode combineert een reeks invoerelementen in een enkelvoudig resultaat door de operator herhaald toe te passen
 - Equivalent met:

```
T result = identity;
for (T element : this stream) {
    result = accumulator.apply(result, element);
}
return result;
```
- Overige reduction methodes:
 - OptionalDouble **average**()
 - long **count**() \leftrightarrow mapToLong(e -> 1).sum()
 - double **sum**() \leftrightarrow reduce((a, b) -> a + b)

- **reduce** en **sum**

```
List<Artikel> artikels = Artikels.getArtikels();

// met reduce
Optional<Double> totaal = artikels.stream()
    .map(Artikel::getPrijs)
    .reduce((a, b) -> a + b);
if (totaal.isPresent()) {
    System.out.println(totaal.get());
}

// alternatief met sum
double som = artikels.stream()
    .mapToDouble(Artikel::getPrijs)
    .sum();
System.out.println(som);
```



- **average**

```
List<Artikel> artikels = Artikels.getArtikels();  
  
OptionalDouble gemiddelde = artikels.stream()  
    .mapToDouble(Artikel::getPrijs)  
    .average();  
System.out.println(gemiddelde.getAsDouble());
```

- **reduce** en **count**

```
List<Artikel> artikels = Artikels.getArtikels();

// met reduce (sum)
long aantalArtikels = artikels.stream()
    .mapToLong(e -> 1)
    .sum();
System.out.println(aantalArtikels);

// alternatief met count
long aantal = artikels.stream()
    .count();
System.out.println(aantal);
```

Typische stream-strukturen

myList

```
.stream()  
.filter (Predicate)  
.limit (long)  
.sorted (Comparator)  
.forEach (Consumer);
```

Set newSet = myList

```
.stream()  
.filter (Predicate)  
.sorted (Comparator)  
.map (Function)  
.collect (Collector);
```

double result = myList

```
.stream()  
.filter (Predicate)  
.map (Function)  
.reduce (BinaryOperator);
```



Opdrachten

- Groeiproject

- module 6
(deel 3: "Streams")



- Opdrachten op BB

- Opgave EmployeeStream
- Opgave Acteur

