

Java

2023 – 2024 : Les 4

jeroen.devos01@ap.be

Lesdoelen

- Lambda
- Optional
- Method reference
- Stream





Lambda (java 8)

- Een lambda is een klein stuk code dat parameters kan aanvaarden en een waarde teruggeeft.
- Kan vergeleken worden met een methode zonder naam.
- Functional programming (denk ook aan JavaScript)

Syntax :

The diagram illustrates the syntax of a Java 8 lambda expression: `(int arg1, String arg2) -> {System.out.println("Two arguments "+arg1+" and "+arg2);}`. It uses horizontal lines and vertical markers to identify the components: the `(int arg1, String arg2)` part is labeled 'Argument List', the `->` is labeled 'Arrow token', and the `{System.out.println("Two arguments "+arg1+" and "+arg2);}` part is labeled 'Body of lambda expression'.

```
(int arg1, String arg2) -> {System.out.println("Two arguments "+arg1+" and "+arg2);}
```

Argument List Arrow token Body of lambda expression

Lambda syntax

- Parameters zijn optioneel
- Bij één parameter mogen de haken weggelaten worden
- Als het parameter type duidelijk is, mag het type weggelaten worden
- Als body één regel is, dan mogen de accolades weggelaten worden
- Als body één regel is, dan mag de return weggelaten worden
- Return is optioneel

```
() -> System.out.println("Hello Lambda");
```

```
(name) -> System.out.println("Hello " + name);
```

```
name -> System.out.println("Hello " + name);
```

```
(first, second) -> first + second;
```

```
(Integer first, Integer second) -> first + second;
```

```
(first, second) -> {  
    System.out.println(first);  
    System.out.println(second);  
    return first + second;  
}
```


Lambda gebruiken

... als parameter

Voorwaarde : de methode heeft als input de Consumer interface

```
List<String> trooperNames = new ArrayList<>();  
trooperNames.add("FN-2198");  
trooperNames.add("FN-2199");  
trooperNames.add("FN-2200");  
  
trooperNames.forEach(s -> System.out.println(s));
```

trooperNames.forEach|

 forEach(Consumer<? super String> action) void

Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards [Next Tip](#)

Consumer interface

```
Consumer<String> action = s -> System.out.println(s);  
  
trooperNames.forEach(action);
```

- Functional interface
- Een Consumer variabele maken:
 - Declaratie : generic van het type dat je verwacht
 - Initialisatie : een lambda functie
- Heeft één method 'accept' : deze voert de lambda uit

Consumer accept

```
List<String> trooperNames = new ArrayList<>();  
trooperNames.add("FN-2198");  
trooperNames.add("FN-2199");  
trooperNames.add("FN-2200");  
  
Consumer<String> action = s -> System.out.println(s);  
  
for (String name : trooperNames) {  
    action.accept(name);  
}
```

De accept methode van de consumer voert de lambda uit.

Consumer als method parameter

```
private void showName(String name, Consumer<String> action) {  
    action.accept(name);  
}  
  
Consumer<String> actionDefault = s -> System.out.println(s);  
Consumer<String> actionSpecial = s -> System.out.println("Hello " + s);  
  
for (String name : trooperNames) {  
    showName(name, actionDefault);  
    showName(name, actionSpecial);  
}
```

Method parameter bepaalt welke lambda er wordt uitgevoerd



Method reference

```
trooperNames.forEach(s -> System.out.println(s));
```

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

- Method reference = verkorte notatie voor de lambda functie
- Syntax : **Object :: methodName**
- **Tip** : IntelliJ zal (meestal) zelf voorstellen om een lambda om te zetten naar een method reference.

Next level sorteren

```
List<StormTrooper> troopers = new ArrayList<>();
troopers.add(new StormTrooper("FN-2198", Rank.SERGEANT));
troopers.add(new StormTrooper("FN-2199", Rank.TROOPER));
troopers.add(new StormTrooper("FN-2200", Rank.TROOPER));

troopers.sort(new SortByNameDescending());
```

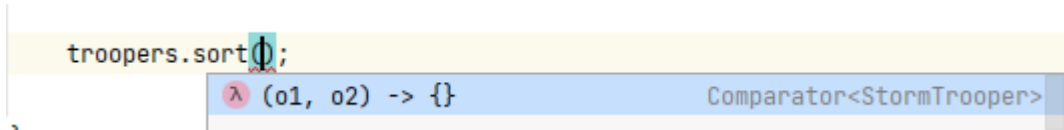
```
public class SortByNameDescending implements Comparator<StormTrooper> {

    public int compare(StormTrooper o1, StormTrooper o2) {
        return o2.getName().compareTo(o1.getName());
    }

}
```

Next level sorteren – stap 2

```
List<StormTrooper> troopers = new ArrayList<>();  
troopers.add(new StormTrooper("FN-2198", Rank.SERGEANT));  
troopers.add(new StormTrooper("FN-2199", Rank.TROOPER));  
troopers.add(new StormTrooper("FN-2200", Rank.TROOPER));  
  
troopers.sort((o1, o2) -> o1.getName().compareTo(o2.getName()));
```



The screenshot shows an IDE with a yellow background. The code `troopers.sort()` is highlighted. A tooltip is visible, showing a lambda expression `(o1, o2) -> {}` and the text `Comparator<StormTrooper>`. The lambda expression is enclosed in a blue box, and the text `Comparator<StormTrooper>` is in a light blue box.

Next level sorteren – stap 3

```
List<StormTrooper> troopers = new ArrayList<>();  
troopers.add(new StormTrooper("FN-2198", Rank.SERGEANT));  
troopers.add(new StormTrooper("FN-2199", Rank.TROOPER));  
troopers.add(new StormTrooper("FN-2200", Rank.TROOPER));  
  
troopers.sort(Comperator.comparing(trooper -> trooper.getName()));
```

```
troopers.sort(Comparator.comparing());
```

```
λ stormTrooper -> {}
```

```
Function<StormTrooper, Object>
```

Weet je nog : Comperator.naturalOrder() ?

Next level sorteren – stap 4

```
List<StormTrooper> troopers = new ArrayList<>();  
troopers.add(new StormTrooper("FN-2198", Rank.SERGEANT));  
troopers.add(new StormTrooper("FN-2199", Rank.TROOPER));  
troopers.add(new StormTrooper("FN-2200", Rank.TROOPER));  
  
troopers.sort(Comperator.comparing(StormTrooper::getName));
```

Vervang de lambda door method reference

Deze notatie wordt in hedendaagse code gebruikt !

Optional<?>



Optional - probleemstelling

```
Ship ship = new Ship();  
Pilot pilot = ship.getPilot();  
  
if(pilot != null) {  
    System.out.println(pilot.toString());  
}
```

- Een object heeft een waarde of is null
- Probleem : elke keer een object wordt gebruikt moet je zeker zijn dat het object niet null is
 - Heb je de code zelf onder controle, dan ben je meestal zeker
 - Komt het object als return value uit een ander object ... dan weet je het niet zeker en moet je controleren

Optional

```
Ship ship = new Ship();  
Optional<Pilot> pilot = ship.getPilot();  
  
if(pilot.isPresent()) {  
    System.out.println(pilot.get().toString());  
}
```

- Optional mag niet null zijn
- Optional heeft een generic als data type
- .isPresent() : zit er een waarde in de Optional (true / false)
- .get() : neem de waarde van de Optional, dit heeft het type van de generic

Optional met lambda

```
Ship ship = new Ship();  
Optional<Pilot> pilot = ship.getPilot();  
  
pilot.ifPresent(value -> System.out.println(pilot.toString()));
```

- .ifPresent() is een kortere schrijfwijze voor de isPresent()/get() constructie en gebruikt een lambda functie

Optional met lambda – deel 2

```
Ship ship = new Ship();  
Optional<Pilot> pilot = ship.getPilot();  
  
pilot.ifPresentOrElse(value -> System.out.println(pilot.toString(),  
    () -> System.out.println("There is no pilot."));
```

- `.ifPresentOrElse()` gebruikt 2 lambda functies
 - De eerste functie : als er inhoud is, lambda heeft één argument
 - De tweede functie : als er geen inhoud is, lambda heeft géén argument

Optional – extra operaties

```
Ship ship = new Ship();  
Optional<Pilot> pilot = ship.getPilot();
```

```
pilot.isEmpty();  
pilot.orElse(...);  
pilot.orElseThrow();
```

- `.isEmpty()` = alternatief van `.isPresent()`
- `.orElse(...)` = zelfde als `.get()`, maar je kan een alternatief meegeven voor het geval er geen waarde is. Opletten met null!
- `.orElseThrow()` = zelfde als `.get()`, maar zal een Exception gooien als er geen waarde is.

Optional – zelf aanmaken

```
Optional<Pilot> existingPilot = Optional.of(new Pilot());  
Optional<Pilot> emptyPilot = Optional.empty();
```

- `Optional.of(...)` : maak een `Optional` van het object aan
- `Optional.empty()` : maak een lege `Optional` aan.

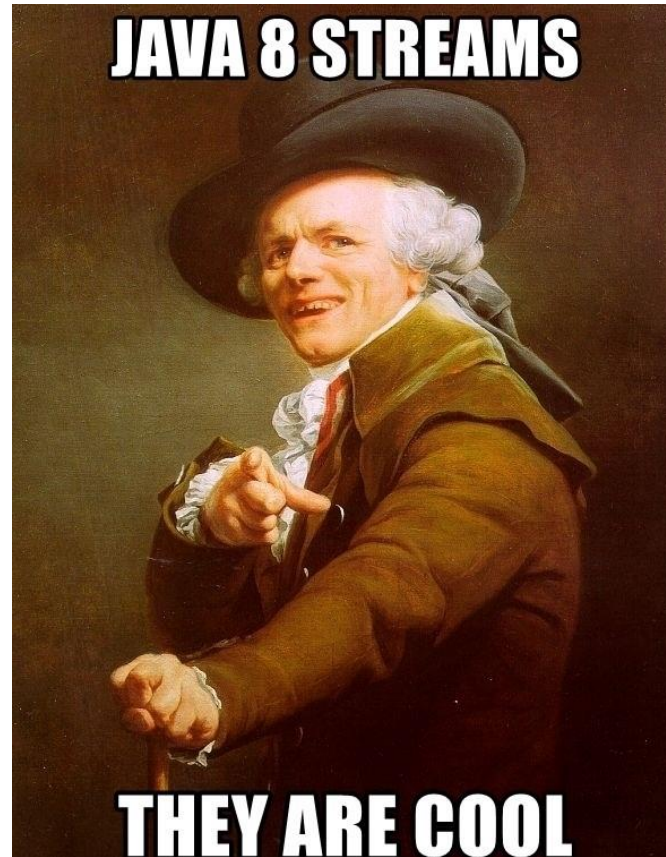
Optional gebruik

Gebruik een Optional enkel :

- als variabele in een methode
- als return type van een methode

Gebruik een Optional **nooit** als method parameter !!

Reden: over een method parameter heb je geen controle. Er bestaat de mogelijkheid dat de gebruiker van de methode als parameter null meegeeft. Je zal dan een null check moeten doen op je Optional, wat ineens het hele nut van de Optional teniet doet.



java.util.stream API

- Wordt gebruikt om Collections te verwerken
- Input: Collection of Array of I/O
- De input wordt niet gewijzigd door de stream
 - Merk op : als we zonder stream een lijst sorteren wijzigt de input wél
- Elke stream operatie geeft een stream terug
 - Functioneel programmeren
- Stream operaties gebruiken lambda's en optionals

Code together

1. Download van digitap de week 4 demo bestanden
2. Zet een nieuw project op
3. Plaats de bestanden in `src/main/java` van je nieuwe project

stream API activeren

```
List<StarWarsMovies> movies = new ArrayList();  
... movies toevoegen ...  
  
movies.stream();
```

- `.stream()` activeert de stream API.
- dit geeft een Stream object terug
 - een Stream object hebben we vrijwel nooit impliciet nodig, we zullen steeds functioneel verder bouwen op de `stream()`.

stream collect operation

```
List<StarWarsMovie> result = movies.stream().collect(Collectors.toList());
```

- .collect() verzamelt het resultaat van de stream in een Collection
- dit is een **copy** van de originele data
- return type is List of Set (of Collection)

```
movies.stream().collect;
```

```
(m) collect(Collector<? super StarWarsMovie, A, R> collector) R
(m) collect(Supplier<R> supplier, BiConsumer<R, ? super StarWarsMovie> accumulator, BiConsumer<R, R> combiner) R
(m) collect(Collectors.toCollection()) Collection<StarWarsMovie>
(m) collect(Collectors.toList()) List<StarWarsMovie>
(m) collect(Collectors.toSet()) Set<StarWarsMovie>
(m) collect(Collectors.toUnmodifiableList()) List<StarWarsMovie>
(m) collect(Collectors.toUnmodifiableSet()) Set<StarWarsMovie>
Press Enter to insert, Tab to replace
```

stream filter operation

```
List<StarWarsMovie> result = movies.stream()  
    .filter(movie -> movie.getEpisode() < 7)  
    .collect(Collectors.toList());
```

- `.filter()` verwijdert objecten uit de stream die niet voldoen aan de criteria
- gebruikt lambda of Predicate om criteria te bepalen
- return type = Stream

```
List<StarWarsMovie> result = movies.stream()  
    .filter()  
    .col λ starWarsMovie -> {} Predicate<StarWarsMovie>
```

Intermezzo : Predicate interface

```
Predicate<StarWarsMovie> episodeFilter = movie -> movie.getEpisode() < 7;
```

```
List<StarWarsMovie> result = movies.stream()  
    .filter(episodeFilter)  
    .collect(Collectors.toList());
```

- Functional interface
- Een Predicate variabele maken:
 - Declaratie : generic van het type dat je wilt filteren
 - Initialisatie : een lambda functie
- Heeft één method 'test' : deze voert de lambda uit
- Of wordt als argument van de .filter stream operation gebruikt

stream sorted operation

```
List<StarWarsMovie> result = movies.stream()  
    .sorted()  
    .collect(Collectors.toList());
```

- `.sorted()` sorteert objecten in de stream volgens de voorziene Comparator
- geen parameter in `sorted()` -> default sortering
- wel parameter : Comparator, lambda of Comparing met method reference
- return type = Stream

```
List<StarWarsMovie> result = movies.stream()  
    .sorted( $\phi$ )  
    .collect(Collectors.toList(),  $\lambda$  (o1, o2) -> {}, Comparator<StarWarsMovie>)
```

stream map operation

```
List<String> result = movies.stream()  
    .map(movie -> movie.getTitle()) //of .map(StarWarsMovie::getTitle())  
    .collect(Collectors.toList());
```

- .map() past de objecten in de stream aan
- gebruikt een lambda functie om de mapping te beschrijven
- HET OBJECT IN DE STREAM KAN VAN DATA TYPE WIJZIGEN !!
- return type = Stream

```
List<StarWarsMovie> result = movies.stream() Stream<StarWarsMovie>  
    .map() Stream<Object>  
    . { starWarsMovie -> {} } Function<StarWarsMovie, Object>
```


stream pipe results

```
List<String> result = movies.stream()  
    .filter(movie -> movie.getEpisode() < 7)  
    .sorted(Comparator.comparing(StarWarsMovie::getTitle))  
    .map(StarWarsMovie::getTitle())  
    .collect(Collectors.toList());
```

Aangezien elke operatie een Stream object als return type heeft, kan je meerdere operaties na elkaar uitvoeren → pipelining

Tip: plaats bij streaming alle operaties onder elkaar!
→ Leesbaarheid én minder conflicten bij VCS (bv GIT)

stream flatMap operation

```
List<StarWarsCharacter> result = movies.stream()  
    .flatMap(movie -> movie.getMainCharacters().stream())  
    .collect(Collectors.toList());
```

- .flatMap() maakt van meerdere streams één stream
- gebruikt een lambda functie om de streams te verzamelen
- HET OBJECT IN DE STREAM KAN VAN DATA TYPE WIJZIGEN !!
- return type = Stream

```
List<StarWarsCharacter> result = movies.stream() Stream<StarWarsMovie>  
    .flatMap() Stream<Object>  
    .coll λ starWarsMovie -> {} Function<StarWarsMovie, Stream<?>>
```

stream distinct operation

```
List<StarWarsCharacter> result = movies.stream()  
    .flatMap(movie -> movie.getMainCharacters().stream())  
    .distinct()  
    .collect(Collectors.toList());
```

- .distinct() haalt duplicaten eruit
- gebruikt de equals() van het object
- return type = Stream

stream forEach operation

```
movies.stream()  
  .flatMap(movie -> movie.getMainCharacters().stream())  
  .distinct()  
  .forEach(character -> System.out.println(character)); // of .forEach(System.out::println);
```

- `.forEach()` past een lambda functie toe op elk element van de stream
- **return type = void**

```
movies.stream() Stream<StarWarsMovie>  
  .flatMap(movie -> movie.getMainCharacters().stream()) Stream<StarWarsCharacter>  
  .distinct()  
  .forEach();  
    λ starWarsCharacter -> {} Consumer<StarWarsCharacter>
```

stream findFirst operation

```
Optional<StarWarsMovie> movie = movies.stream()  
    .findFirst();
```

- `.findFirst()` neemt het eerste element van de stream
- kan null zijn -> `Optional`!
- return type = `Optional` van data type van de stream

Tip: gebruik in combinatie met de filter operatie

stream findAny operation

```
Optional<StarWarsMovie> movie = movies.stream()  
    .findAny();
```

- .findAny() neemt een willekeurig element van de stream
- kan null zijn -> Optional!
- return type = Optional van data type van de stream

Tip: gebruik voor een stabiel resultaat findFirst en vermijd findAny

stream orElse operation

```
StarWarsMovie movie = movies.stream()  
    .findFirst()  
    .orElse(null);
```

```
StarWarsMovie movie = movies.stream()  
    .findAny()  
    .orElse(null);
```

- .orElse() geeft je de mogelijkheid om een waarde mee te geven voor het geval er géén elementen in de stream zitten
- return type = data type van de stream (**geen Optional meer**)

stream matching operations

```
boolean match = movies.stream()  
    .anyMatch(movie -> movie.getEpisode() == 1);
```

```
boolean match = movies.stream()  
    .allMatch(movie -> movie.getEpisode() == 1);
```

```
boolean match = movies.stream()  
    .noneMatch(movie -> movie.getEpisode() == 1);
```

- Matching vergelijkt de elementen in de stream met een lambda functie
- return type = boolean

ALMOST THERE



WE ARE

makeameme.org

Streams and numbers

- Om met getallen te werken zijn er drie specifieke streams voorzien:
 - IntStream
 - LongStream
 - DoubleStream
- Deze streams hebben géén generic, het data type zit er vast in.
- Ze hebben dezelfde operaties als een gewone Stream + extra's

stream mapToInt operation

```
IntStream stream = movies.stream()  
    .mapToInt(movie -> movie.getEpisode()); // of .mapToInt(StarWarsMovie::getEpisode);
```

- .mapToInt() maakt van een Stream een IntStream
- de lambda functie beschrijft de conversie naar int
- return type = IntStream

Zelfde verhaal voor .mapToLong() en .mapToDouble()

stream sum operation

```
int sum = movies.stream()  
    .mapToInt(StarWarsMovie::getEpisode)  
    .sum();
```

- .sum() telt alle getallen in de stream op
- return type = int

stream count operation

```
long count = movies.stream()  
    .mapToInt(StarWarsMovie::getEpisode)  
    .count();
```

- `.count()` telt het aantal getallen in de Stream
- return type = long

stream min / max / average operation

```
int min = movies.stream()  
    .mapToInt(StarWarsMovie::getEpisode)  
    .min()  
    .orElse(0);
```

- `.min()` neemt het minimum van de stream
- `.orElse()` bepaalt wat je doet als de stream leeg is
- return type = int

Idem voor `.max()` en `.average()`

stream summaryStatistics operation

```
IntSummaryStatistics statistics = movies.stream()  
    .mapToInt(StarWarsMovie::getEpisode)  
    .summaryStatistics();
```

- .summaryStatistics() berekent statistieken op je Stream van getallen
- return type = IntSummaryStatistics

Het SummaryStatistics object bevat : min, max, average, sum en count.



**FOR TODAY
ANYWAY...**