



# JAVA 8: Lamda en Streams

# Table of contents

- Lamda Expressions
- Functional Interfaces
- Method References
- Data verwerken
- Lamda's in collections
- Streams

# Lamda Expressions

Java 8

# Lamda Expressions

= een anonieme functie.

Voor LAMDA:

```
Comparator<String> comp = new Comparator<String>() {  
    @Override  
    public int compare(String o1, String o2) {  
        return o1.length() - o2.length();  
    }  
};
```

# Lamda Expressions

MET LAMDA:

```
Comparator<String> com = (o1, o2) -> o1.length() - o2.length();
```

# Lamda Expressions

## SYNTAX:

<i>Lijst van argumenten</i>	<i>Een pijl symbool</i>	<i>Body</i>
(int x, int y)	->	x + y

- Lijst van argumenten: parameters
- Pijl: stuurt argumenten naar de body
- Body: ontvangt argumenten en doet hier iets mee

# Functional interfaces

Java 8

# Functional Interface

= interface met slechts 1 abstracte methode!

Alleen deze interfaces kunnen gebruikt worden bij Lamda's.

BV:

```
@FunctionalInterface  
public interface Comparator<T>
```

# Functional Interface

Zelf één maken:

```
@FunctionalInterface  
public interface MyInterface {  
  
    void doeIets();  
  
}
```

→ Annotatie is extra info voor compiler

# Functional Interface

Gebruik van eigen interface:

```
MyInterface i = () -> System.out.println("Hello Functional Interface!");
i.doeIets();
```

# Functional Interface

## Ander voorbeeld:

```
@FunctionalInterface  
public interface MyInterface {  
  
    int geefIetsTerug(String naam);  
  
}  
  
MyInterface i = (String naam) -> naam.length();  
System.out.println("Lengte van de string: " + i.geefIetsTerug("Kenneth"));  
  
MyInterface i = (s) -> s.length();  
System.out.println("Lengte van de string: " + i.geefIetsTerug("Kenneth"));  
  
MyInterface i = (String naam) -> {  
    System.out.println("Naam: " + naam);  
    return naam.length();  
};  
System.out.println(i.geefIetsTerug("Kenneth"));
```

# Method References

Java 8

# Method References

- Statische methodereferentie
- Gebonden methodereferentie
- Ongebonden methodereferentie
- Constructoreferentie

# Statische methodereferentie

```
@FunctionalInterface  
public interface MyInterface {  
  
    int geefIetsTerug(String getal);  
}
```

```
MyInterface i = Integer::parseInt;  
System.out.println(i.geefIetsTerug("100"));
```

# Ongebonden methodereferentie

```
@FunctionalInterface  
public interface MyInterface {  
  
    String geefIetsTerug(String naam);  
  
}  
  
MyInterface i = String::toUpperCase;  
System.out.println(i.geefIetsTerug("Kenneth"));
```

# Gebonden methodereferentie

```
@FunctionalInterface  
public interface MyInterface {  
  
    int geefIetsTerug();  
  
}  
  
String hello = "Hello World!";  
  
MyInterface i = hello::length;  
System.out.println(i.geefIetsTerug());
```

# Constructorreferentie

```
@FunctionalInterface  
public interface MyInterface {  
  
    String geefIetsTerug();  
}
```

```
MyInterface i = String::new;
```

# Samenvatting:

<b>Soort referentie</b>	<b>Syntax</b>
Statische methodereferentie	ClassName::staticMethodName
Gebonden methodereferentie	objectName::methodName
Ongebonden methodereferentie	ParameterType::methodName
Constructorreferentie	Classname::new

# Data Verwerken

Java 8

# Consumer

Consumeert een object en doet niets terug.

## accept

```
void accept(T t)
```

Performs this operation on the given argument.

**Parameters:**

t - the input argument

Bv:

```
Consumer<String> printer = s -> System.out.println(s);  
printer.accept("Hello World!");
```

```
Consumer<String> printer2 = System.out::println;  
printer2.accept("Hello World!");
```

# Supplier

Geeft een object terug, verwacht niets:

get

T get()

Gets a result.

Returns:

a result

Bv:

```
Supplier<String> object = () -> new String();
System.out.println(object.get().concat("Hello"));

Supplier<String> object2 = String::new;
System.out.println(object2.get().concat("World"));
```

# Function

Verwacht een object en geeft een object van ander of zelfde type terug:

## apply

R `apply(T t)`

Applies this function to the given argument.

**Parameters:**

`t` - the function argument

**Returns:**

the function result

Bv:

```
Function<String, Integer> functie = string -> string.length();
System.out.println(functie.apply("Hello World!"));
```

```
Function<String, Integer> functie2 = String::length;
System.out.println(functie2.apply("Hello World!"));
```

# Predicate

Verwacht een object en geeft een boolean terug

## test

boolean test(T t)

Evaluates this predicate on the given argument.

**Parameters:**

t - the input argument

**Returns:**

true if the input argument matches the predicate, otherwise false

Bv:

```
Predicate<String> test = string -> string.isEmpty();
System.out.println(test.test("Hello World!"));
Predicate<String> test2 = String::isEmpty;
System.out.println(test2.test(new String()));
```

# Collections

Java 8

# foreach

**Collection<? extends String> c**

```
List<String> list = new ArrayList<>();  
list.addAll(Arrays.asList("groet", "hallo", "hello", "hey", "goedendag"));  
list.for
```

**forEach(Consumer<? super String> action) : void - Iterable**

@ Java  
Methods

Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception. Unless otherwise specified by the implementing class, actions are performed in the order of iteration (if an iteration order is specified). Exceptions thrown by the action are relayed to the caller.

**Parameters:**

**action** The action to be performed for each element

**Throws:**

[NullPointerException](#) - if the specified action is null

**Since:**

1.8

**@implSpec**

The default implementation behaves as if

Press 'Tab' from proposal table or click for focus

Press 'Ctrl+Space' to show Template Proposals

# foreach: list

```
List<String> list = new ArrayList<>();  
list.addAll(Arrays.asList("groet", "hallo", "hello", "hey", "goedendag"));  
list.forEach(System.out::println);
```

→ Uitprinten van de volledige lijst!

# sort

```
List<String> list = new ArrayList<>();  
list.addAll(Arrays.asList("groet", "hallo", "hello", "hey", "goedendag"));  
list.sort(Comparator.comparingInt(String::length));
```

→ Sorteren van de lijst op lengte van de strings.

```
List<String> list = new ArrayList<>();  
list.addAll(Arrays.asList("groet", "hallo", "hello", "hey", "goedendag"));  
list.sort(Comparator.comparingInt(String::length).thenComparing(String::compareTo));
```

→ Als de lengte hetzelfde is, sorteren op alfabet.

# foreach: map

```
Map<Integer, String> map = new HashMap<>();  
map.put(1, "groet");  
map.put(2, "hallo");  
map.put(3, "hello");  
map.put(4, "hey");  
map.put(5, "goedendag");  
map.forEach((key, value) -> System.out.println(key + ": " + value));
```

→ Uitprinten van de volledige map!

# Streams

Java 8

# Streams

= Pijplijn, waar objecten doorheen doorstromen.

- In de pijplijn worden er vaak tussenliggende bewerkingen gedaan.
- Op het einde is er altijd een afsluitende bewerking.

# Soorten streams

- 1) Stream <T>
- 2) IntStream
- 3) LongStream
- 4) DoubleStream

Bv:

```
IntStream.rangeClosed(0, 10).forEach(System.out::println);
```

```
Stream.of(" Mijn String ")  
    .map(String::toUpperCase)  
    .map(String::trim)  
    .forEach(System.out::println);
```

# Streams & Collections

Veelgebruikte methodes bij streams:

- **Stream:** starten van de stream
- **Map:** verander het type van de stream  
→ Verwacht een Function als parameter
- **Filter:** uitfilteren van data uit de stream  
→ Verwacht een Predicate als parameter

# Voorbeeld

```
List<String> list = new ArrayList<>();  
list.addAll(Arrays.asList("groet", "hallo", "hello", "hey", "goedendag"));  
list.stream().map(String::toUpperCase).filter(s -> !s.isEmpty()).forEach(System.out::println);
```

**stream** → opent de stroom

**map** → Plaatst alle strings in hoofdletters

**filter** → Alleen deze strings die niet leeg zijn

**foreach** → Uitprinten van de strings

**Resultaat:**

```
GROET  
HALLO  
HELLO  
HEY  
GOEDENDAG
```

# Ander voorbeeld

```
List<String> list = new ArrayList<>();  
list.addAll(Arrays.asList("groet", "hallo", "hello", "hey", "goedendag"));  
list.stream().mapToInt(String::length).filter(lengte -> lengte > 5).forEach(System.out::println);
```

**stream** → opent de stroom

**mapToInt** → Verandert de lijst naar integer waardes

**filter** → Alleen deze strings met een lengte groter dan 5

**foreach** → Uitprinten van de lengtes

**Resultaat:** 9

# Terminal vs Intermediate

## Terminal methode:

Beëindigt de stroom en voert deze uit

Bv: foreach

## Intermediate methode:

Geeft een nieuwe stroom in de plaats

Bv: map