HO
GENT

OOSDII - Lambda Expressies

Table of Contents

1. Doelstellingen

Na het studeren en maken van de oefeningen van dit hoofdstuk ben je in staat om volgende zaken te herkennen, toe te lichten, te definiëren, toe te passen en te implementeren:

- Anonieme innerklasse
- Lambda expressies
- Methode referenties
- Functionele interface
- De basis functionele interfaces waaronder Function, Predicate, Consumer, Supplier …

2. Inleiding

Een functionele interface is een interface met één abstracte methode.

Lambda expressies laten toe om in éénzelfde stap zowel de implementatie van een functionele interface te definiëren als een instantie van deze implementatie te creëren.

Lambda expressies implementeren de abstracte methode van een functionele interface en zodus ook de functionele interface zelf.

Lambda expressies laten volgende functionaliteit toe:

- De gecreëerde instantie laat toe om functionaliteit te behandelen als een methode argument, functionaliteit wordt behandeld als data die je kan doorgeven.
- Een functie implementeren zonder dat deze toebehoort aan een klasse.
- Een lambda expressie kan aanzien worden als een object (het creëert een object), de bijhorende functionaliteit kan op vraag uitgevoerd worden.

Java zet met Lambda Expressies een stap in de richting van **functioneel programmeren**, een specifieke vorm van declaratief programmeren:

- Bij declaratief programmeren wordt source code zodanig geschreven dat deze meer aangeeft wat men van de geschreven code verwacht, met weinig of geen nadruk op de eigenlijke implementatie. Je geeft aan ('declareert') wat je wil doen, niet hoe het gedaan wordt.
- Imperatief programmeren is de tegenhanger van declaratief programmeren. Als je bij declaratief programmeren net aangeeft wat je wil bereiken, kan je imperatief programmeren beschouwen als het lijn per lijn instructies coderen om te bereiken wat moet gedaan worden.

- Declaratief programmeren: "Wat wil ik bereiken"
- Imperatief programmeren: "Hoe wil ik het bereiken"

In dit hoofdstuk ligt de focus op "Lambda Expressies". In een later hoofdstuk gaan we deze lambda expressies gebruiken in de context van "Functioneel Programmeren".

3. Implementatie van een interface

Tot op heden gingen we een interface altijd implementeren in een aparte klasse. Een interface kan je ook implementeren in een geneste klasse, lokale klasse of anonieme klasse.

In onderstaand voorbeeld wordt de interface `HelloWorld` gedeclareerd. Deze interface wordt binnen de methode `sayhello` zowel geïmplementeerd als lokale klasse (`EnglishGreeting`) en als anonieme klasse (`FrenchGreeting`).

```
1  public class HelloWorldLocalClass {
2
3      @FunctionalInterface
4      private interface HelloWorld { ①
5          public void greetSomeone(String someone);
6      }
7
8      public void sayHello() {
9          // Local class
10         class EnglishGreeting implements HelloWorld { ②
11             @Override
12             public void greetSomeone(String someone) {
13                 System.out.println("Hello " + someone);
14             }
15         }
16
17         HelloWorld englishGreeting = new EnglishGreeting(); ③
18
19         // Anonymous class
20         HelloWorld frenchGreeting = new HelloWorld() { ④
21             @Override
22             public void greetSomeone(String someone) {
23                 System.out.println("Salut " + someone);
24             }
25         };
26
27         englishGreeting.greetSomeone("world"); ⑤
28         frenchGreeting.greetSomeone("Fred");
29     }
30
31     public static void main(String[] args) {
32         HelloWorldLocalClass myApp =
33             new HelloWorldLocalClass();
34         myApp.sayHello();
35     }
36 }
```

① Declaratie van de interface `HelloWorld`

② Declaratie van de lokale klasse `EnglishGreeting` die de interface `HelloWorld` implementeert.

③ Instantiëren van een EnglishGreeting object.

④ Instantiëren van een HelloWorld object, waarvan de implementatie werd gedeclareerd via een anonieme klasse.

⑤ Gebruik van de objecten.

Een interface kan je implementeren in een lokale klasse, een klasse gedeclareerd binnen een code blok, bv. binnen een methode. Zo een lokale klasse heeft als voordeel dat ze enkel daar gekend is bij naam (in scope) en enkel binnen dat blok kan gebruikt worden.

Je kan zelfs een stap verder gaan en een interface implementeren in een anonieme klasse. Dit laat je toe om in één stap een klasse te declareren en te instantiëren. Een anonieme klasse is als een lokale klasse, maar dan zonder naam. Een anonieme klasse ga je meestal declareren op de plaats waar je ze nodig hebt en gebruik je als je ze slechts éénmalig nodig hebt.

Anonieme klassen worden meestal gebruikt in situaties waarin u een lichtgewicht klasse moet kunnen maken om als parameter te worden doorgegeven. (Zie later vb. Comparator.)

Met een anonieme klasse declareer en instantieer je een klasse in één stap.

Een anonieme klasse is als lokale klasse, maar dan zonder naam.

4. Anonieme innerklasse

4.1. Declaratie

Een lokale klasse is een klasse declaratie, een anonieme inner klasse is een expressie. De definitie van een anonieme inner klasse gebeurt dus in de expressie zelf. De expressie laat je toe om in één stap een klasse te declareren en te instantiëren.

In vorig voorbeeld werd een anonieme inner klasse gebruikt binnen het initialisatie statement van de lokale variabele `frenchGreeting`. Een lokale klasse werd gebruikt voor de initialisatie van de lokale variabele `englishGreeting`.

Met een anonieme inner klasse kan je een interface implementeren en instantiëren in de expressie, alsook een subklasse van een abstracte of concrete klasse.

4.2. Syntax

Een anonieme inner klasse is een expressie!

De syntax van een anonieme inner klasse expressie is analoog aan het aanroepen van een constructor, behalve dat deze constructor vervangen wordt door een klasse definitie binnen een code blok.

```
1        HelloWorld frenchGreeting = new HelloWorld() { ④
2            @Override
3            public void greetSomeone(String someone) {
4                System.out.println("Salut " + someone);
5            }
6        };
```

De expressie van de anonieme inner klasse bevat het volgende:

- de `new` operator
- De naam van de te implementeren interface of de uit te breiden klasse. In bovenstaand voorbeeld gaat de anonieme inner klasse de interface `HelloWorld` implementeren.
- De haakjes omvatten de argumenten van de constructor, zoals bij de creatie van een normale klasse object. Let op: als enkel de methodes van een interface worden geïmplementeerd, dan heb je voldoende aan een default constructor. De haakjes zullen dan leeg zijn zoals in het voorbeeld.
- Een klasse declaratie met daarin de declaratie en implementatie van de methodes.

Een anonieme inner klasse definitie is een expressie, die deel moet uitmaken van een statement.

5. Functionele interfaces

Een functionele interface is een interface met exact één abstracte methode.

De abstracte methode in een functionele interface kan je aanzien als een contract van een methode handtekening die je later met een lambda expressie kan implementeren.

Voorbeeld van een functionele interface:

Module java.base
Package java.util

Interface Comparator<T>

Type Parameters:

T - the type of objects that may be compared by this comparator

All Known Implementing Classes:

Collator, RuleBasedCollator

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

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

Method Summary

All Methods	Static Methods	Instance Methods	Abstract Methods	Default Methods

Modifier and Type	Method	Description
int	compare(T o1, T o2)	Compares its two arguments for order.

compare is de enige echte abstracte methode van Comparator (equals overschrijft een public methode uit Object, en daarvoor telt dit niet als een abstracte methode)

6. Lambda Expressies

Ook een lambda expressie is een expressie die je zoals anonieme klassen toelaat om in één stap een functionele interface te declareren en te instantiëren.

Met een lambda expressie kan je enkel een functionele interface implementeren en instantiëren in de expressie. Een lambda expressie wordt vaak gebruikt om zo een functionele interface te implementeren en te instantiëren en die instantie mee te geven als argument aan een methode. Een functionele interface is een interface met exact één abstracte methode.

Lambda expressies laten ons toe functionaliteit door te geven als methode argument, functionaliteit wordt zo behandeld als data die je kan doorgeven.

```
1  public class HelloWorldLambdaExpression {
2
3      @FunctionalInterface ①
4      interface HelloWorld {
5          public void greetSomeone(String someone);
6      }
7
8      public void sayHello() {
9
10         HelloWorld dutchGreeting = (String someone) -> { ②
11             System.out.println("Hallo " + someone);
12         };
13
14         dutchGreeting.greetSomeone("Pete");
15     }
16
17     public static void main(String[] args) {
18         HelloWorldLambdaExpression myApp =
19             new HelloWorldLambdaExpression();
20         myApp.sayHello();
21     }
22 }
```

① Lambda expressies kunnen enkel gebruikt worden om functionele interfaces te implementeren. De annotatie geeft aan dat het gaat om een functionele interface, deze is net zoals bij de annotatie @Override niet verplicht.

② De lambda expressie voorziet in de implementatie van de functionele interface en creëert er in dezelfde stap ook een instantie van.

6.1. Syntax van Lambda Expressies

Een lambda expressie bevat volgende onderdelen:

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

- **parameterlijst**: een parameterlijst ingesloten in haakjes

Een formele parameterlijst is een lijst van parameters zoals je deze hebt bij de declaratie van een methode: het zijn de parameters die de methode verwacht. Bij een lambda expressie zijn dit de parameters die de lambda expressie verwacht.

Java beschikt over 'variable type inference' waardoor de compiler vaak uit de context het type van de parameters kan afleiden. Je kan dit type dus meestal weglaten bij de parameters. Als er maar één parameter is mag je ook de haakjes weglaten.

- Het pijltje hoort bij de syntax van de lambda expressie: →
- Een body, die kan bestaan uit één enkele expressie of uit een statement blok. Als je één enkele expressie specifieert zonder statement blok, dan zal de Java runtime deze expressie evalueren en het resultaat teruggeven.

Onderstaande lambda expressie is equivalent aan voorgaand voorbeeld:

```
1 HelloWorld dutchGreeting = someone -> System.out.println("Hallo " + someone);
```

7. Methode Referenties

Aangezien lambda expressies enkel van toepassing zijn op functionele interfaces kan je ze ook aanzien als anonieme methodes, de implementatie van die ene abstracte methode uit de functionele interface.

In sommige gevallen doet een lambda expressie ook niet meer dan het aanroepen van een bestaande methode. In deze gevallen wordt de code leesbaarder door te refereren naar die methode die de lambda expressie aanroept.

Methode referenties laten je toe om te refereren naar een bestaande methode gebruik makende van zijn naam. Op die manier zijn methode referenties compacte en eenvoudig leesbare lambda expressies voor methodes die al een naam hebben.

```
1  public class HelloWorldMethodReference {
2
3      @FunctionalInterface
4      interface HelloWorld {
5          public void greetSomeone(String someone);
6      }
7
8      public void sayHello() {
9          HelloWorld dutchGreeting = HelloWorldMethodReference::printGreeting; // <1>
10         dutchGreeting.greetSomeone("Pete");
11     }
12
13     private static void printGreeting(String name) { ②
14         System.out.println("Hallo " + name);
15     }
16
17     public static void main(String[] args) {
18         HelloWorldMethodReference myApp =
19             new HelloWorldMethodReference();
20         myApp.sayHello();
21     }
22 }
```

① Een lambda expressie die gebruikt maakt van een methode referentie

② De methode waarnaar gerefereerd wordt.

Merk op dat het return type en parameterlijst van de referentie methode exact gelijk moet zijn aan deze van de abstracte methode gedeclareerd in de functionele interface!

7.1. Soorten Methode Referenties

Er bestaan 4 soorten methode referenties:

Soort	Voorbeeld
Een referentie naar een klasse methode (`static` methode)	ContainingClass::staticMethodName

Een referentie naar een instantie methode van een specifiek object. De instantie methode van dat object zal aangeroepen worden, het lambda argument wordt als argument doorgegeven.	containingObject::instanceMethodName
Een referentie naar een instantie methode van een arbitrair object van een specifiek type. De instantie methode zal aangeroepen worden op het lambda argument.	ContainingType::methodName
Een referentie naar een constructor. Dit creëert een lambda die de default constructor van de gespecifieerde klasse aanroept.	ClassName::new

Voorgaand voorbeeld bevat een voorbeeld van een methode referentie naar een klasse methode. De andere soorten komen aan bod in het hoofdstuk rond functioneel programmeren.

8. Basis functionele interfaces

De Java API voorziet ons van enkele basis functionele interfaces:

- vijf ervan werken met een enkel referentie type: Predicate, Unary Operator, Function, Supplier en Consumer
- één ervan werkt met twee referentie types: BinaryOperator

De letters T en R in onderstaand overzicht geven aan dat het gaat om generieke interfaces. Deze letters stellen de types voor die later pas ingevuld worden (net zoals bij het gebruik van een List<T>).

Functionele interface	**Prototype - abstracte methode**	**Beschrijving**
Function<T, R>	R apply(T t)	Het return type is verschillend van het type van de argumenten (één of twee argumenten) (17 variaties)
Predicate<T>	boolean test(T t)	Neemt één of twee argumenten en geeft een boolean terug (5 variaties)
Consumer<T>	void accept(T t)	Neemt één of twee argumenten, het return type is `void` (8 variaties)
Supplier<T>	T get()	Neemt geen argumenten en geeft een waarde terug (5 variaties)
BinaryOperator<T>	T apply(T t1, T t2)	Beide argument types en het return type zijn identiek (4 variaties)

UnaryOperator<T>	T apply(T t)	Het argument type en return type zijn identiek (4 variaties)

Overzicht:

- Elk van deze interfaces heeft 3 varianten die primitieve datatypen aanvaarden: double, int of long
- Volgende variaties aanvaarden ook twee argumenten: BiPredicate, BiFunction, BiConsumer
- Function heeft 6 variaties die primitieve datatypes (double, int en long) omzetten naar andere primitieve datatypes
- Function en BiFunction hebben elk 3 variaties die een referentie type aanvaarden en een primitief datatype teruggeven: double, int of long
- Supplier heeft een variant dat een boolean teruggeeft
- BiConsumer heeft drie variaties die een referentie type aanvaardt en een primitief datatype: double, int of long

BiConsumer<T,U>	void accept(T t, U u)
BiFunction<T,U,R>	R apply(T t, U u)
BiPredicate<T,U>	boolean test(T t, U u)

8.1. Function

De functionele interface 'Function' definieert een prototype van een methode met één enkele parameter en een return type.

```
1 public interface Function<T,R> {
2     <R> apply(T parameter);
3
4     // ...
5 }
```

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

        // Function met een Integer als argument en
        // en Double als return type
        Function<Integer, Double> half = a -> a / 2.0;  ①

        // Voer de Function methode apply uit met argument 10.
        System.out.println(half.apply(10));

        // Maak een samengestelde Function die eerst halveert
        // en dan verdriedubbelt.
        half = half.andThen(a -> 3 * a);    ②

        // Voer de samengestelde Function uit met argument 10
        System.out.println(half.apply(10));
    }
}
```

① Lambda expressie: implementatie van de abstracte methode in de interface Function en het instantiëren van een object

② Gebruik van de default methode **andThen**, die een samenstelling maakt van de vorige lambda expressie (en deze eerst uitvoert) en deze nieuwe implementatie (die als tweede wordt uitgevoerd op het resultaat van de eerste)

Een andere default methode is de methode **identity**. Deze methode retourneert een Function die steeds zijn argument retourneert.

```java
    static <T> Function<T, T> identity() {
        return t -> t;
    }
```

8.2. Predicate

Voorziet een methode die een boolean waarde teruggeeft, gebaseerd op één argument.

```java
public interface Predicate<T> {
    boolean test(T t);

    // ...
}
```

```java
public class PredicateTest {
    public static void pred(int number, Predicate<Integer> predicate)
    {
        // Voer de Predicate uit op het eerste argument number
        if (predicate.test(number)) {
            System.out.printf("De voorwaarde op nummer %d is waar.%n", number);
        } else {
            System.out.printf("De voorwaarde op nummer %d is fout.%n", number);
        }
    }

    public static void main(String[] args)
    {
        // Predicate met voorwaarde "< 18"
        Predicate<Integer> lessThan18 = i -> (i < 18);  ①

        // Test de voorwaarde
        System.out.println(lessThan18.test(10));

        // Predicate met voorwaarde "> 12"
        Predicate<Integer> greaterThan12 = (i) -> i > 5;

        // Test de voorwaarde
        System.out.println(greaterThan12.test(10));

        // Samengestelde predicate
        Predicate<Integer> lessThan18AndGreaterThen12 = lessThan18.and
(greaterThen12);

        boolean result = lessThan18AndGreaterThen12.test(16); ②
        System.out.println(result);

        // Negatie van een Predicate
        boolean result2 = lessThan18AndGreaterThan12.negate().test(16); ③
        System.out.println(result2);

        //passing Predicate into function
        pred(10, (i) -> i > 7);

        // OR Predicate
        Predicate<String> hasLengthOf10 = t -> t.length() > 10;
        Predicate<String> containsLetterA = p -> p.contains("A");
        String containsA = "And";

        boolean outcome = hasLengthOf10.or(containsLetterA).test(containsA); ④
        System.out.println(outcome);
    }
}
```

① Lambda expressie: implementatie van de abstracte methode in de interface Predicate en het

instantiëren van een object

② De default methode **and** geeft een samengestelde Predicate terug, een short-circuit logische AND operatie van beide implementaties

③ De default methode **negate** geeft een Predicate terug, de negatie van de reeds bestaande Predicate.

④ De default methode **or** geeft een samengestelde Predicate terug, een short-circuit logische OF van beide implementaties.

8.3. Consumer

Voorziet een methode die één argument aanvaardt, zonder return waarde.

```
1 public interface Consumer<T> {
2     void accept(T t);
3
4     // ...
5 }
```

```java
public class ConsumerAccept {
    public static void main(String args[])
    {
        // Consumer to display a number
        Consumer<Integer> display = a -> System.out.println(a); ①

        // Implement display using accept()
        display.accept(10);

        List<Integer> list = new ArrayList<Integer>();
        list.add(2);
        list.add(1);
        list.add(3);

        // Consumer to display a number
        Consumer<List<Integer>> displayList = a -> System.out.println(a);

        // Implement display using accept()
        displayList.accept(list);

        // Consumer die elk element in een lijst verdubbelt
        Consumer<List<Integer>> multiplyElementsByTwo = a -> {
            for (int i = 0; i < a.size(); i++)
                a.set(i, 2 * a.get(i));
        };

        // Samengestelde Consumer: verdubbel elk element in de lijst en
        // druk elk element af op het scherm
        addTen.andThen(displayList).accept(list); ②
    }
}
```

① Lambda expressie: implementatie van de abstracte methode in de interface Predicate en het instantiëren van een object

② De default methode **andThen** geeft een samengestelde Consumer terug die achtereenvolgens de originele consumer implementatie uitvoert gevolgd door deze implementatie.

8.4. Supplier

Voorziet een methode die een object teruggeeft, dit kan een nieuw geïnstantieerd object zijn.

```java
public interface Supplier<T> {
    T get();

    // ...
}
```

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

        // This function returns a random value.
        Supplier<Double> randomValue = Math::random;  // OF () -> Math.random();

        // Print the random value using get()
        System.out.println(randomValue.get());
    }
}
```

8.5. BinaryOperator

Voorziet in een methode met twee argumenten en een returnwaarde van hetzelfde type.

BinaryOperator is een speciaal geval van de functionele interface BiFunction, waar argumenten en return waarde van hetzelfde type zijn.

8.6. UnaryOperator

Voorziet in een methode met één argument en een returnwaarde van hetzelfde type.

UnaryOperator is een speciaal geval van de functionele interface Function, waar argumenten en returnwaarde van hetzelfde type zijn.

9. Referenties

- https://docs.oracle.com/javase/tutorial/
- https://dzone.com/articles/cheatsheet-java-functional-interfaces
- http://tutorials.jenkov.com/java-functional-programming/functional-interfaces.html
- https://www.geeksforgeeks.org